A Goal Model for Adaptive Complex Systems

Scott A. DeLoach, Matthew Miller
Department of Computing & Information Sciences
Kansas State University
234 Nichols Hall, Manhattan, KS USA 66506
(785) 532-6350
{sdeloach, millermj}@ksu.edu

Abstract *– Goal based systems have seen increasing interest in complex, adaptive systems. While there are a number of approaches to eliciting goal-based requirements and to using goals at runtime, there are no frameworks that use goals for requirements while providing a direct mapping to goals used to drive the system at runtime. The Goal Model for Dynamic Systems (GMoDS) presented in this paper allows a designer to specify goals during requirements and then use those same goals throughout system development and at runtime.*

1 Introduction

Software technology has evolved rapidly over the last few decades. As early software systems typically solved small problems in a single domain, much of a programmer's time was spent making algorithms run efficiently within memory and processing constraints. As a result, the abstraction level of the design was not far above to that of code. As processing speed and memory size has increased, the expectations of the software running on those machines have increased as well. Specifically, users now expect their systems to be "intelligent" enough to adapt to the problem being solved and to the environment within which the system is executing. To meet these expectations, software designers have attempted to create software that emulates human problem solving abilities using information from a variety of resources covering multiple domains. While this approach has yielded increasingly capable systems, system complexity has increased dramatically making it more difficult to understand, analyze, and build such systems.

Goal models have been proposed as an approach to managing software system complexity by helping to manage requirement complexity and to build systems that can adapt more readily to changes in requirements [12]. Goals capture "why" a particular task is required. Thus, goals are the key to building systems that can adapt to a wide variety of situations including changes in the environment, changes in capabilities of system components, and changes in the problem to be solved [2]. In a distributed system, it is often the case that several tasks can be performed by several different system components. Only by knowing the goal (or the reason why) the task needs to be performed can the system make an intelligent choice about which task to execute on which component.

Systems that use goals at runtime have been a staple in the Artificial Intelligence (AI) planning community for decades [5]. AI planners allow systems to choose the appropriate steps required to achieve a specified state of the world. Planning algorithms recursively decompose this system level goal into sub-goals until actions are found that achieve those sub-goals. Other architectures use reactive planning such as the Procedural Reasoning System (PRS) of Georgeff and Lansky [5]. While similar to the goal model proposed in this paper, these approaches are generally limited to single agents and do not provide support from requirements through implementation.

Thus, we have found no frameworks for developing complex systems that use goals to capture requirements while providing a direct mapping to goals that drive the system at runtime. The benefits of such a framework would be twofold. First, such a framework would provide a consistent view of the requirements from analysis, through design to implementation, thus ensuring that requirements were directly accounted for at all stages of the system development. Second, such a framework would allow the system to adapt to the dynamics of the problem being solved as well as the system's environment.

In this paper, we present our Goal Model for Dynamic Systems (GMoDS), which provides a set of models for capturing system level goals and *for using those goals during design and at runtime* to allow the system to adapt to dynamic problems and environments. GMoDS has three distinct models: a goal specification model that captures the designer intent, a runtime model that defines the semantics of the specification model, and an execution model that implements the instance model. Next, we review related work, followed by the definition of the GMoDS models. We conclude with an example of their use and a discussion of results and future work.

1.1 Related Work

Goals have been used to capture requirements for traditional systems as well as agent-oriented systems as goals tend to capture "what" the system is supposed to do instead of "how" the system is supposed to behave. It has been argued that goals are a more natural way to model system requirements, as they tend to change less often that the functions of the software.

The Knowledge Acquisition in autOmated Specification (KAOS) framework was developed for modeling system requirements [12]. The KAOS model is closely related to requirements elicitation and allows system goals to be specified at a high-level and then decomposed (either disjunctively or conjunctively) into a set of sub-goals. KAOS also specifies that goals can contribute to or degrade the achievement of other goals and allows the system to determine which agents in the system are best capable of achieving specific goals.

The i* framework was designed to for organization based systems [14] and focuses on the interactions between actors, which are considered to be autonomous. Actors may control limited resources forcing interaction between actors to accomplish system goals. Dependencies between the actors are modeled in a strategic dependency model, which specifies interactions between actors. The i* framework focuses on early requirements elicitation and guiding system design; it is not an implementation device.

On the other end of the spectrum, the PRACTIONIST framework [9] includes the notion of a runtime goal model to support the implementation of individual BDI agents. In the PRACTIONIST framework, desires and intentions are derived from the system goals captured in the goal model. Relationships between goals include inconsistency, entailment, precondition, and depends. However, the PRACTIONIST framework assumes agents are designed using traditional BDI approaches and therefore is not generalizable.

Multiagent system specification is not a trivial task and thus there has been a significant amount of work toward making the specification of multiagent systems a sound and complete process. Many agent-oriented methodologies use goals explicitly either at design time or runtime [10], [7], [11], [6], and [3] Goal models have been used for requirements elicitation in many agent-based methodologies including Gaia, MaSE, ROADMAP, and Tropos; however, none use goals directly during implementation or runtime.

2 GMoDS Definition

Due to space limitations of this paper, we cannot provide the complete formal semantics of the GMoDS model. For a more complete definition of the semantics, the reader is referred to [8].

2.1 Specification Model

The specification model has three main entities: goals, events, and parameters. A *goal* is an observable desired state of the world while an *event* is an observable phenomenon that occurs during system execution. Goal and event *parameters* provide information to agents on the details of the goal to be achieved or the specific event occurrence. While an agent is pursuing a goal, a number of events may occur. These events may cause other goals in the system to be added or removed. The new goals added to the system are parameterized based on the parameters of the event that triggered their creation. Essentially, an agent can specialize its performance based on the parameters of the goal it has been assigned to achieve, which, in turn, are based on the parameters of the triggering event. GMoDS defines two types of goals: goal classes and goal instances. *Goal classes* are goals that are specified by the system designer to model the goal interactions within the organization. *Goal instances* are the runtime instantiation of a goal class with specific parameters. The *instance/instanceOf* relations capture the connection between goal classes and goal instances. Each goal instance is an instance of exactly one goal class, and a goal class may have zero or more goal instances.

2.1.1 Goal Specification Tree

Goal trees are a natural approach to problem decomposition as shown Figure 1. Upper level goals (*parents*) are decomposed into lower level sub-goals (*children*) and each parent has either a conjunctive or disjunctive achievement condition as shown via the «and» and «or» decoration in Figure 1. Goals without children are known as *leaf goals*. A GMoDS *goal specification tree* (G_{Spec}) specifies how the goal classes are related to one another. The goal classes in the goal specification tree are analogous to the specification of classes in an object-oriented language. Classes are designed before hand, and then are dynamically instantiated during runtime, each instance having its own set of attributes. The instances of a goal class are independent of each other. The goal instances are inserted into a *goal instance tree* ($G_{Instance}$) during runtime. Each goal instance is achieved independently of every other instance of that goal. For example, suppose that a goal class exists to rescue a victim. If there are three instances of the rescue goal, they would represent three victims needing to be rescued and the achievement of each rescue goal would be independent.

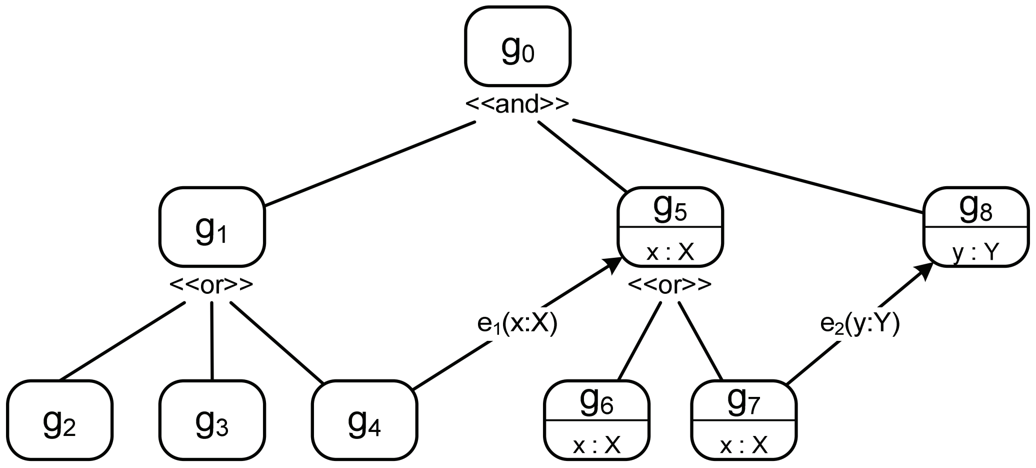

Figure 1. Goal specification tree

2.1.2 Goal Triggers

GMoDS uses a set of relations within the tree structure to specify how runtime goals may interact. Because goal instances are created based on the occurrence of specific events, the effect of these events on $G_{Instance}$ must be specified. The *triggers* relation between g_1 and g_2 predicated on event e_1 specifies that a new goal instance of class g_2 is created when e_1 occurs during the pursuit of a goal instance of class g_1. Likewise, the *negative trigger* relation from g_1 to g_2 on event e_1 specifies that goal g_2 should be removed from $G_{Instance}$ when e_1 occurs. Figure 1 includes a trigger relation from g_4 to g_5 and from g_7 to g_8. However, triggers cannot be used randomly. Suppose the designer placed a trigger with event e_3 between g_4 and g_6 in Figure 1. If event e_3 occurs before e_1, no instance of g_5 would exist and thus g_6 would have no parent goal. Obviously, this is illegal; all inbound triggers must come from the children of the same sub-tree.

To bootstrap $G_{Instance}$, we define an *initial event* (or *initial trigger*), e_0, which must occur to add an initial set of goals (including the root goal) to $G_{Instance}$. When the system starts, the initial event occurs and the root goal is added to $G_{Instance}$. Then all children not triggered by some other event are systematically and recursively added to $G_{Instance}$.

2.1.3 Goal Precedence

To allow a full or partial ordered execution of goals in the system, the designer may specify *goal precedence* in the goal specification via the *precedes* relation. Goal precedence ensures that no agents work on a specific goal until all goals that precede that goal have been achieved. An example would be that of object identification and manipulation. The object must be first identified before being manipulated.

Figure 2 is an extension of the example in Figure 1 with precedence relations inserted between g_2 and g_3 and between g_6 and g_7. Because the notion of precedence only applies to goals in the same triggered subtree, not all instances of g_6 must be achieved before any goals of g_7 may be pursued. Precedence only requires that all instances of g_6 in the same subtree (under goal g_5) must be achieved before instances of g_7 in the same subtree may be pursued.

There are several restrictions on specifying goal precedence in G_{Spec}. The first restriction is that of precedence cycles; clearly, if a precedence cycle exists then none of the goals may ever be pursued. Additionally, a cycle of mixed triggers and precedes relationships is disallowed since the mixed cycle also creates a set of goals that cannot be assigned. It is also obvious that a goal should never precede any of its ancestors (parents, grandparents, etc.) A goal that preceded its parents would imply that the parent must be achieved before the child could be pursued.

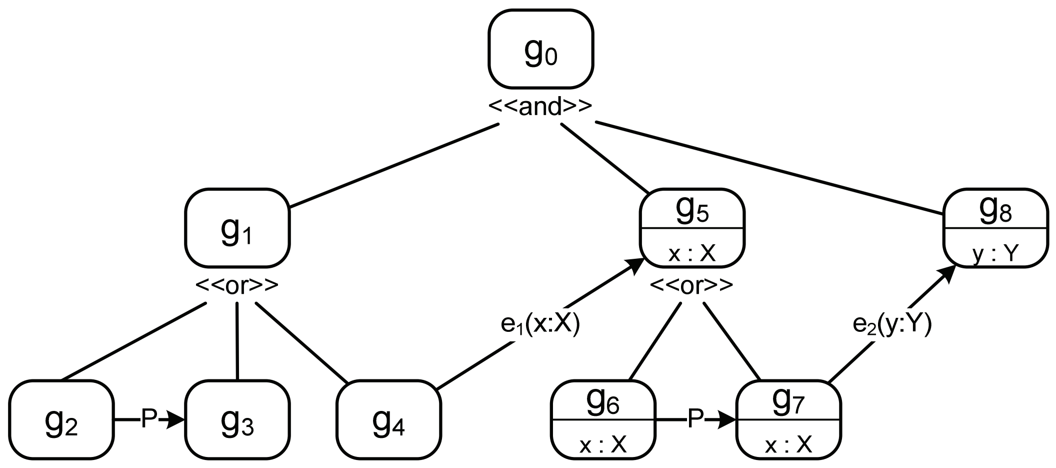

Figure 2. Specification tree with precedence

1.1.1 G_{Spec} Formalization

Formally, the GMoDS specification tree, G_{Spec}, is defined as a tuple <G_s, E_s, subgoal, triggers, ¬triggers, precedes> where

```
GS                  set(GoalClass)
ES                  set(EventClass)

GoalClass           ⟨name, FormalParameters⟩
EventClass          ⟨name, FormalParameters⟩
FormalParameters    ⟨set(ParameterType)⟩
ParameterType       ⟨name, type⟩
```

The basic tree structure, precedence, and triggers are captured by a set of relations over goal classes and event classes.

```
subgoal       ⊆ GS × GS
triggers      ⊆ GS × ES × GS
¬triggers     ⊆ GS × ES × GS
precedes      ⊆ GS × GS
```

The constraints on G_{Spec} with in relation to definition as a tree are captured in the definitions and constraints presented in Table 1.

There are several constraints from the discussion above about how precedence and triggers can and cannot be used (see Table 2). However, to be able to specify these constraints, we need additional formal relations based on G_{Spec}, namely the closure of the precedence and triggers relations. The precedence closure includes all goals directly preceded as well as indirectly preceded via a chain of precedence relations. In addition, the children of all preceded goals are in the precedence closure. The formal recursive definition is given below.

$$
\begin{aligned}
&\forall g, g_1, g2 : G_S \\
&\qquad (g, g_1) \in \text{precedes} \Rightarrow (g, g_1) \in \text{precedes}^+ \\
&\qquad (g, g_1) \in \text{precedes}^+ \wedge (g_1, g_2) \in \text{subgoal} \Rightarrow (g, g_2) \in \text{precedes}^+ \\
&\qquad (g, g_1) \in \text{precedes}^+ \wedge (g_1, g_2) \in \text{precedes} \Rightarrow (g, g_2) \in \text{precedes}^+
\end{aligned}
$$

The triggers closure is defined in a similar manner to the precedence closure. The formal recursive definition of triggers closure is given below (the $\neg\text{triggers}^+$ relation is defined in a similar manner.).

$$
\begin{aligned}
&\forall g, g_1, g_2 : G_S \\
&\qquad (g, g_1) \in \text{triggers} \Rightarrow (g, g_1) \in \text{triggers}^+ \\
&\qquad (g, g_1) \in \text{triggers}^+ \wedge (g_1, g_2) \in \text{subgoal} \Rightarrow (g, g_2) \in \text{triggers}^+ \\
&\qquad (g, g_1) \in \text{triggers}^+ \wedge (g_1, g_2) \in \text{triggers} \Rightarrow (g, g_2) \in \text{triggers}^+
\end{aligned}
$$

One other constraint of interest relates to ensuring that cycles of triggers and precedence relations are not allowed. To check this constraint, we need to examine the closure of the composition of the triggers and precedes relations, which is defined as $(\text{triggers} \circ \text{precedes})^+$. Any goal related to itself the closure

Table 1. Definitions and Constraints on G_{Spec}

Definitions	
$\text{root} : G_S \rightarrow \text{Boolean}$	True if goal g is root goal of G_{Spec}
$\text{conjunctive} : G_S \rightarrow \text{Boolean}$	True if goal g is conjunctive parent goal
$\text{disjunctive} : G_S \rightarrow \text{Boolean}$	True if goal g is disjunctive parent goal
$\text{leaf} : G_S \rightarrow \text{Boolean}$	True if goal g is a leaf goal
Constraints	
$\forall g : G_S\ (g, g) \notin \text{subgoal}^+$	Goal g cannot be its own descendant (no loops)
$\exists! g : G_S\ \text{root}(g) = \text{True}$	There is exactly one root goal
$\forall g : G_S\ \neg\text{root}(g) \Rightarrow \exists! g_1 : G_s\ (g_1, g) \in \text{subgoal}$	All goals have exactly 1 parent except the root
$\forall g : G_S\ \text{root}(g) \Rightarrow \neg\exists g_1 : G_s\ (g_1, g) \in \text{subgoal}$	The root goal has no parents
$\forall g : G_S\ (\exists g_1 : G_S\ (g, g_1) \in \text{subgoal}) \Rightarrow \neg\text{leaf}(g) \wedge (\text{conjunctive}(g) \otimes \text{disjunctive}(g))$	Non-leaf goals are conjunctive or disjunctive
$\forall g : G_S\ (\neg\exists g_1 : G_S\ (g, g_1) \in \text{subgoal}) \Rightarrow \text{leaf}(g) \wedge \neg(\text{conjunctive}(g) \vee \text{disjunctive}(g))$	Leaf goals are neither conjunctive or disjunctive

of the composition is either in a precedence cycle, a triggers cycle, or in a mixed precedence/triggers cycle, all of which are illegal.

Table 2. Additional Constraints on G_{Spec}

Constraints	
$\forall g_1, g_2, g_3, g_4 : G_S\ (g_1, g_2) \in triggers \wedge (g_3, g_2) \in subgoal^+ \wedge (g_4, g_3) \in triggers \Rightarrow (g_4, g_2) \in subgoal^+$	All inbound triggers must come from the children of the same sub-tree
$\forall\ g : G_S\ (g, g) \notin (triggers \circ precedes)^+.$	Cycles of triggers and precedes relationships are not allowed
$\forall g_1, g_2 : G_S\ (g_1, g_2) \in precedes^+ \Rightarrow ((g_1, g_2) \notin subgoal^+ \wedge (g_2, g_1) \notin subgoal^+)$	Goals may not precede any of its ancestors or descendants

2.2 Runtime Model

G_{Spec} allows the designer to specify the system hierarchy and the relations between goal classes. G_{Spec} is used as a template to create $G_{Instance}$ at runtime. $G_{Instance}$ retains the structure of G_{Spec} while allowing dynamism by way of triggering and precedence. At runtime, an instance goal is created from a goal specification by creating an instance goal with the same name and parameter types as its associated specification goal. A goal's parameter values are provided by the triggering event or are inherited from its parent in G_I. The subgoal^I relation is defined by the system as it creates goal instances and is s analogous to its counterpart in G_S.

G_I	set(GoalInstance)
GoalInstance	$\langle$name, ActualParameters$\rangle$
ActualParameters	$\langle$set(Parameter)$\rangle$
Parameter	$\langle$name, type, value$\rangle$
subgoal^I	$\subseteq G_I \times G_I$

To track the state of the system, the predicates *achieved, obviated*, *preceded*, and *failed* are dynamically set for each goal in $G_{Instance}$.

preceded :	$G_I \rightarrow$ Boolean
obviated :	$G_I \rightarrow$ Boolean
achieved :	$G_I \rightarrow$ Boolean
failed :	$G_I \rightarrow$ Boolean

The *achieved* predicate states whether a goal has been achieved by the system. For leaf goals, *achieved* becomes True when the agent pursing the goal notifies the system of its achievement. For parent goals, the value of the achieved predicate is based on the achievement condition (conjunction or disjunction) and the state of its children. The *obviated* predicate states whether a goal is no longer needed by the system. A goal becomes obviated if it is a child of a disjunctive goal that has been achieved that does not precede any other system goal. The *preceded* predicate is True if a goal preceding it has not been achieved or if a new goal may still be instantiated that may precede it. The *failed* predicate is True if the system has deemed that the goal can never be achieved by the system.

2.3 Execution Model

The Execution Model implements GMoDS as defined above using a collection of goal sets, which are analyzed for completeness and correctness in [8]. The definition of the GMoDS Execution Model makes the implementation of GMoDS straightforward. The semantics of the GMoDS Execution Model is based on set theory, in which G_I is partitioned into six sub-sets: $G_{I\text{-Triggered}}$, $G_{I\text{-Active}}$, $G_{I\text{-Achieved}}$, $G_{I\text{-Removed}}$, $G_{I\text{-Failed}}$ and $G_{I\text{-Obviated}}$ as shown in Figure 3. Membership in these sets is based upon the set of axioms defined in [8] such that the sets partition G_I. The arrows in Figure 3 indicate allowable transitions of goals between sets.

The first set is the *triggered set* $G_{I\text{-Triggered}}$, were all goals are placed upon instantiation. Goals stay in this set until one of the other predicates (obviated, preceded, failed, or achieved) become True. The *active set*, $G_{I\text{-Active}}$, is the set of goals that are triggered but not preceded. Essentially, $G_{I\text{-Active}}$ is the set of goals that the system may pursue. Goals in $G_{I\text{-Active}}$ remain there until they are achieved, failed, removed, or obviated. When an agent achieves a goal, that goal is moved from the $G_{I\text{-Active}}$ into $G_{I\text{-Achieved}}$. Once goals have moved into the $G_{I\text{-Achieved}}$, they cannot move to any other set aside from $G_{I\text{-Removed}}$. The *removed set* ($G_{I\text{-Removed}}$) contains goals that have been removed as the result of a negative trigger. A goal in the removed set cannot be moved to any other set. When a goal is removed, it is treated as if it never existed in the system, which means that any precedence/triggers relations related to that goal cease to exist when that goal is removed. The *failed set* ($G_{I\text{-Failed}}$) contains goals that the system can never achieve. Once a goal has been placed in $G_{I\text{-Failed}}$, it goal may never leave as achievement implies that the system has completed the goal, while removal makes it as if the goal never existed. The *obviated set* ($G_{I\text{-Obviated}}$)

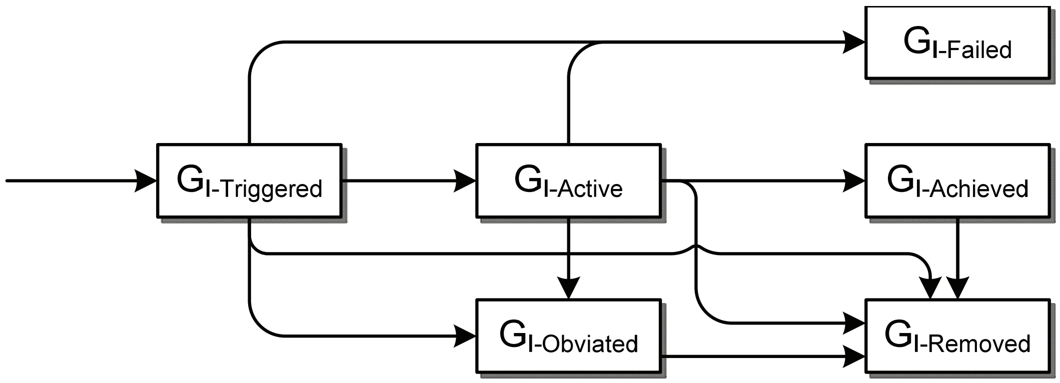

Figure 3. Execution Model

contains goals that no longer need be pursued by the system. These goals are not achieved, and should not be assigned to any agents. Goals in $G_{I\text{-Obviated}}$ may be removed, but not failed. A formal definition on the constraints imposed by the execution model is listed below.

$$\begin{aligned}
&\forall g{:}G_I\; g \in G_{I\text{-Triggered}} && \Leftrightarrow \neg obviated(g) \wedge \neg achieved(g) \wedge preceded(g) \wedge \neg removed(g) \\
& && \quad \wedge \neg failed(g) \\
&\forall g{:}G_I\; g \in G_{I\text{-Active}} && \Leftrightarrow \neg obviated(g) \wedge \neg achieved(g) \wedge \neg preceded(g) \wedge \neg failed(g) \\
&\forall g{:}G_I\; g \in G_{I\text{-Achieved}} && \Leftrightarrow achieved(g) \wedge \neg preceded(g) \wedge \neg\, removed(g) \wedge \neg\, failed(g) \\
&\forall g{:}G_I\; g \in G_{I\text{-Failed}} && \Leftrightarrow failed(g) \wedge \neg\, removed(g) \wedge \neg achieved(g) \\
&\forall g{:}G_I\; g \in G_{I\text{-Removed}} && \Leftrightarrow removed(g) \wedge \neg\, failed(g) \wedge \neg achieved(g) \\
&\forall g{:}G_I\; g \in G_{I\text{-Obviated}} && \Leftrightarrow obviated(g) \wedge \neg\, removed(g) \wedge \neg\, failed(g)
\end{aligned}$$

2.3.1 Runtime Execution Module

The system interacts with the runtime execution module via two operations: *occurred*, and *initialTrigger*. The *initialTrigger* operation creates the initial instance of G_I, including all goals instantiated by the initial trigger e_0. The *occurred* operation updates G_I based on the occurrence of any other event. There are two types of events of interest: application specific events as defined in the G_{Spec}, and general events such as goal achievement or goal failure. The runtime execution model modifies G_I appropriately based on the event that occurred. When goals are achieved, the runtime execution module updates G_I by moving the appropriate goals into $G_{I\text{-Achieved}}$, moving goals from $G_{I\text{-Triggered}}$ into $G_{I\text{-Active}}$ and moving goals to $G_{I\text{-Obviated}}$. Then, all goals in $G_{I\text{-Triggered}}$ are checked to see if their precedence restrictions have been removed. If all precedence restrictions are satisfied, the goals are moved to $G_{I\text{-Active}}$.

2.3.2 Execution Example

For example, if Figure 2 is our specification tree, then as the system starts the initial event (e_0) trigger occurs and all goals that are not explicitly triggered in G_{Spec} are instantiated as shown in Figure 4, which represents the goal instance tree. We use parentheses in to the value of the parameter passed from the triggering event.

At this point, $G_I = \{g_0, g_1, g_2, g_3, g_4\}$ with preceded(g_3) = True. We assume all predicates are False unless specifically noted. At this the point, the execution model look as follows.

$$\begin{aligned}
G_I &= \{g_0, g_1, g_2, g_3, g_4\} \\
G_{I\text{-Triggered}} &= \{g_3\} \\
G_{I\text{-Active}} &= \{g_0, g_1, g_2, g_4\} \\
G_{I\text{-Achieved}} &= \{\,\} \\
G_{I\text{-Failed}} &= \{\,\} \\
G_{I\text{-Removed}} &= \{\,\} \\
G_{I\text{-Obviated}} &= \{\,\}
\end{aligned}$$

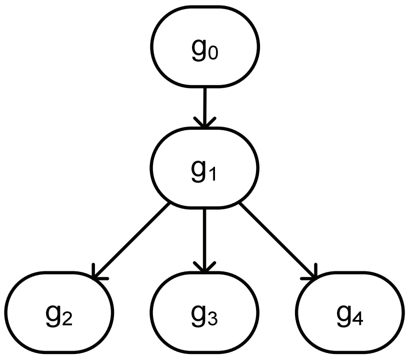

Figure 4. Goal instance tree after initial triggers

Goal precedence restricts the system's choice of goals to pursue. For example, in Figure 4, the system would typically pursue all of the leaf goals in $G_{I\text{-Triggered}}$, if possible. However, due the precedence specified in Figure 2, goal g_3 is not included in $G_{I\text{-Active}}$ and thus may not be pursued. The system is only able to pick goals from $G_{I\text{-Active}}$ to attempt to achieve.

Continuing with our example, if event e_1 (with parameter of x=1) occurs during the pursuit of g_4, goal $g_5(1)$ is instantiated along with its descendents. In fact, multiple instances of e_1 may occur during the pursuit of g_4 as shown Figure 5, which shows two separate occurrences of e_1, the second with parameter x=2.

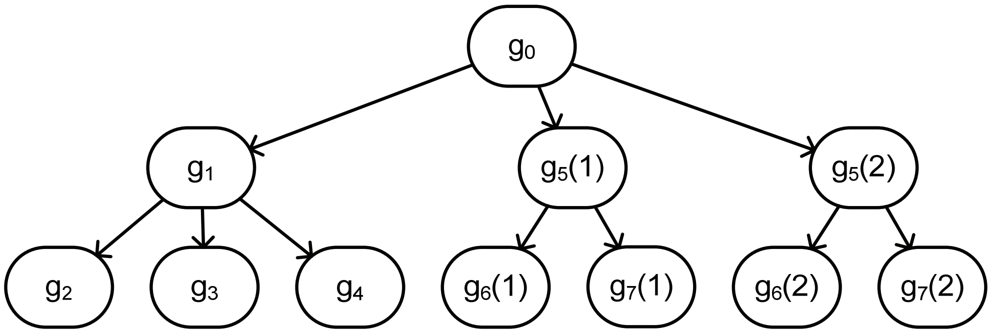

Figure 5. $G_{Instance}$ tree after second e_1 event

At this point, $G_I = \{g_0, g_1, g_2, g_3, g_4, g_5(1), g_6(1), g_7(1), g_5(2), g_6(2), g_7(2)\}$ with *preceded*(g_3) = *preceded*($g_7(1)$) = *preceded*($g_7(2)$) = True. The status of the execution model is given below.

```
G_I             = {g0,g1,g2,g3,g4,g5(1),g6(1),g7(1),g5(2),g6(2),g7(2)}
G_I-Triggered   = {g3, g7(1),g7(2)}
G_I-Active      = {g0,g1,g2,g4,g5(1),g5(2),g6(1),g6(2)}
G_I-Achieved    = {}
G_I-Failed      = {}
G_I-Removed     = {}
G_I-Obviated    = {}
```

Disjunctive goals allow the system to choose which goals to pursue, although there is no constraint on the choice of which disjunctive child goal to pursue. In Figure 5, for example, goals g_2 or g_4 may both be pursued, or the system may choose to pursue just one of them. However, once a disjunctive goal is achieved (e.g. $g_6(2)$ gets achieved thus achieving $g_5(2)$), its non-achieved children (e.g., $g_7(2)$) become obviated. Thus, assuming goal $g_6(1)$ is achieved then *achieved*($g_6(1)$) = True and *achieved*($g_5(1)$) = True, since $g_5(1)$ is disjunctive (i.e., *disjunctive*(*instanceOf*($g_5(1)$)) = True). As described above, goal $g_7(1)$ becomes unnecessary and *obviated*($g_7(1)$) = True. The execution model now looks like the following.

$$\begin{array}{ll}
G_{I} & = \{g_0, g_1, g_2, g_3, g_4, g_5(1), g_6(1), g_7(1), g_5(2), g_6(2), g_7(2)\} \\
G_{I\text{-Triggered}} & = \{g_3, g_7(2)\} \\
G_{I\text{-Active}} & = \{g_0, g_1, g_2, g_4, g_5(2), g_6(2)\} \\
G_{I\text{-Achieved}} & = \{g_5(1), g_6(1)\} \\
G_{I\text{-Failed}} & = \{\} \\
G_{I\text{-Removed}} & = \{\} \\
G_{I\text{-Obviated}} & = \{g_7(1)\}
\end{array}$$

At this point, we assume goal g_4 is achieved resulting in both the achievement of g_1 and the obviation of g_2 and g_3. The new state of the execution models becomes

$$\begin{array}{ll}
G_{I} & = \{g_0, g_1, g_3, g_4, g_5(1), g_6(1), g_7(1), g_5(2), g_6(2), g_7(2)\} \\
G_{I\text{-Triggered}} & = \{g_7(2)\} \\
G_{I\text{-Active}} & = \{g_0, g_5(2), g_6(2)\} \\
G_{I\text{-Achieved}} & = \{g_5(1), g_6(1), g_4, g_1\} \\
G_{I\text{-Failed}} & = \{\} \\
G_{I\text{-Removed}} & = \{\} \\
G_{I\text{-Obviated}} & = \{g_7(1), g_2, g_3\}
\end{array}$$

Finally, if $g_6(2)$ is achieved, then $g_5(2)$ is achieved and $g_7(2)$ is obviated as before. In addition, since g_1, and all instances of g_5 have been achieved (and trivially all instances of g_8), g_0 becomes achieved as well and we get the results shown below.

$$\begin{array}{ll}
G_{I} & = \{g_0, g_1, g_3, g_4, g_5(1), g_6(1), g_7(1), g_5(2), g_6(2), g_7(2)\} \\
G_{I\text{-Triggered}} & = \{\} \\
G_{I\text{-Active}} & = \{\} \\
G_{I\text{-Achieved}} & = \{g_5(1), g_6(1), g_4, g_1, g_6(2), g_5(2), g_0\} \\
G_{I\text{-Failed}} & = \{\} \\
G_{I\text{-Removed}} & = \{\} \\
G_{I\text{-Obviated}} & = \{g_7(1), g_2, g_3, g_7(2)\}
\end{array}$$

As we can see, there are no longer any active or triggered goals; therefore, there can be no more goals instantiated and the system is done. In addition, because all goals are either achieved or obviated, we know that the overall goal of the system has been achieved. In actuality, we could have failed or removed goals and still have the system be successful as long as the overall system goal, g_0, is achieved.

3 Application Example

The application used to demonstrate the runtime execution model is a simulated Weapons of Mass Destruction (WMD) search system, where a team of robots attempts to find, detect, and remove WMDs [8]. In the system, several robots (agents) search an area looking for objects. When objects are found, the robots determine whether the objects are WMDs or inert. The team can detect three weapon types: biological, chemical, and nuclear. Each team has exactly one robot capable of detecting each type. If the object fails a test for one type, it must be tested for the other types. Only if an object fails all three tests can it be classified as inert.

3.1 Goal Specification Tree Design

The goal specification tree (G_{Spec}) for the WMD system is shown in Figure 6. The top-level goal is *WMDSearch*, which has two children, *FindWMD* and *RemoveWMD*. The intent of *FindWMD* is to search an area and identify all objects; the intent of *RemoveWMD* is to remove a verified weapon. *FindWMD* is conjunctively decomposed into four children. The first, *Initialize*, determines the number of robots in the system. The second goal, *AssignArea*, divides the area to be searched into sub-areas. *SearchArea* defines a sub-area to be searched. If objects are found during the search, an *IdentifyObject* goal is triggered, which seeks to identify weapons by their type. The *IdentifyObject* goal has three children: *CheckChem*, *CheckRadio*, and *CheckBio*, whose purpose is to identify specific weapon types.

The triggers relations are denoted in Figure 6 by an arrow with an event name. The goals *Initialize*, *AssignArea*, *SearchArea* and *IdentifyObject* are triggered by different event types: initial, assign, search, and objectFound respectively. When a WMD has been positively identified, a WMDdetected event triggers a *RemoveWMD* goal and the other children of the parent *IdentifyObject* goal are removed from G_I via a negative trigger (identified by a dotted arrow). The *RemoveWMD* goal is preceded by the *IdentifyObject* goal, which is denoted by and arrow with the label «Precedes». Thus, *RemoveWMD* goals cannot be pursued until all *IdentifyObject* goals have been achieved.

The *IdentifyObject* goal is conjunctively decomposed to ensure that all checks are done if no WMDs are detected (the goals are achieved even when the test is run, even if no WMD is found). If decomposed disjunctively, the *IdentifyObject* goal would be achieved as soon as the first check was successfully completed, regardless of whether a WMD was detected. The negative triggers are included to ensure that once a test has identified a WMD, no more tests are run as a *WMDdetected* event triggers a *RemoveWMD* goal.

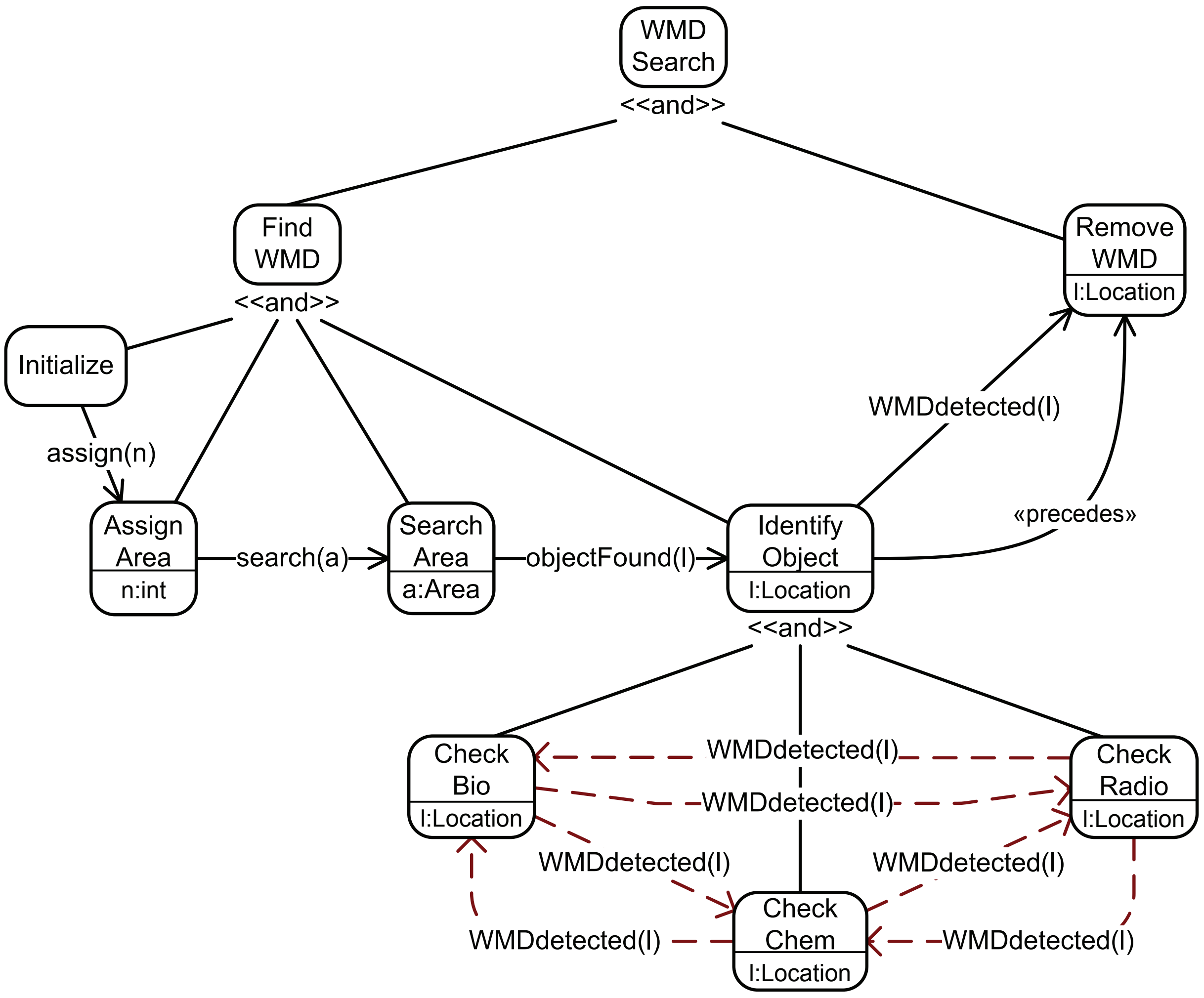

Figure 6. WMD goal specification tree

3.2 Goal Instance Tree

The *WMDSearch* goal instance tree, G_I, is used at runtime to guide the WMD system as it reacts to events. To demonstrate the effectiveness of G_I at runtime, a snapshot was taken each time events occurred that caused a change to G_I. For brevity, we only show a portion of the snapshots. Snapshots depict G_I (goals in $G_{I\text{-Triggered}}$ and $G_{I\text{-Active}}$ only) in tree form to allow comparison of G_I to G_{Spec}. Grey ovals with dashed edges denote the causal events while the parameters for each goal are shown in parenthesis.

Figure 7 and Figure 8 show the operation of the system at startup. Figure 7a depicts G_I after the *initial trigger* while Figure 7b shows the instantiation of an *AssignArea(4)* goal in response to a subsequent *assign* event. Figure 7c represents the state of G_I after the *Initialize* goal is achieved (and moved to $G_{I\text{-Achieved}}$). Figure 8a shows the result of a *search* event, addition of a *SearchArea([0,0],[20,20])* goal, (the parameters indicate the subarea). In Figure 8b, *AssignArea(4)* is achieved.

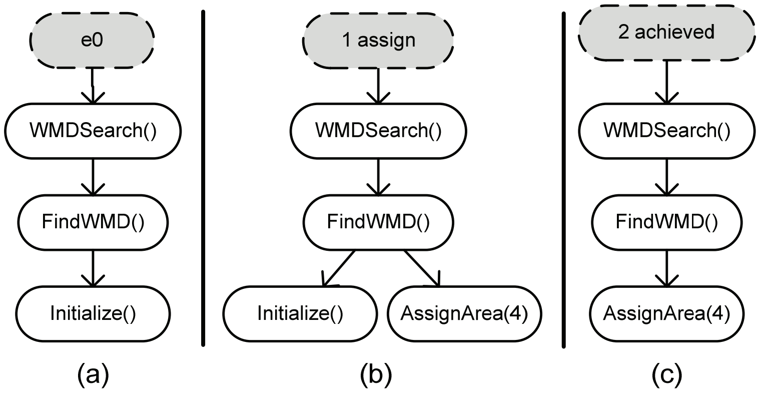

Figure 7. Initial, assign, and achieve events

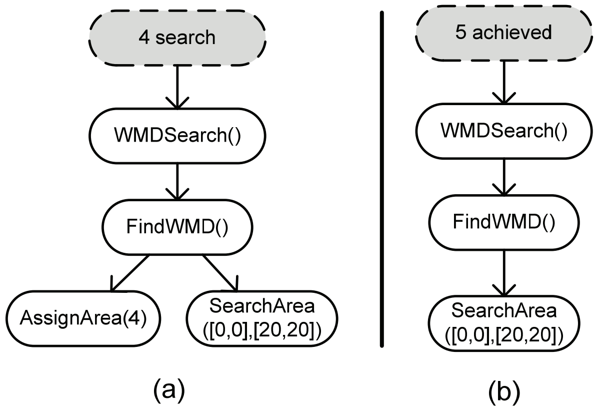

Figure 8. Search and achieve events

When an object is found, an *objectFound* event occurs causing an *IdentifyObject(1,1)* goal and its children *CheckChem(1,1)*, *CheckRadio(1,1)*, and *CheckBio(1,1)* to be created as shown in Figure 9. Figure 10 shows a second occurrence of an *objectFound* event and a second instance of an *IdentifyObject(2,4)* goal, which shows how multiple instances of the same goal are allowed in G_I while still conforming to G_{Spec}.

Next, the *CheckBio(1,1)* goal is achieved resulting in the G_I of Figure 11. Figure 12 shows the state of G_I after a *WMDdetected* event triggers a *RemoveWMD(2,4)* goal followed by the achievement of the *IdentifyObject(2,4)* goal. (The double line around *RemoveWMD(2,4)* indicates that it is preceded and cannot be moved into $G_{I\text{-}Active}$.)

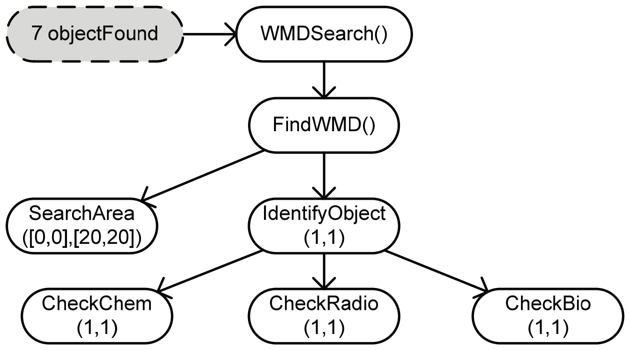

Figure 9. IdentifyObject triggered at 1,1

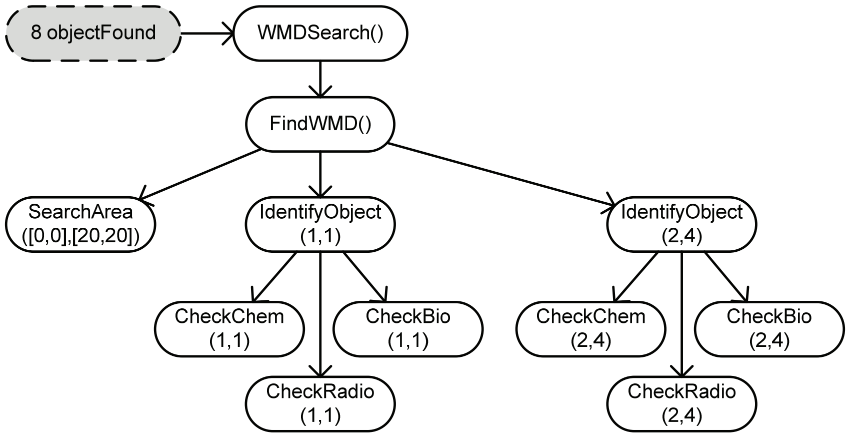

Figure 10. IdentifyObject triggered at 2,4

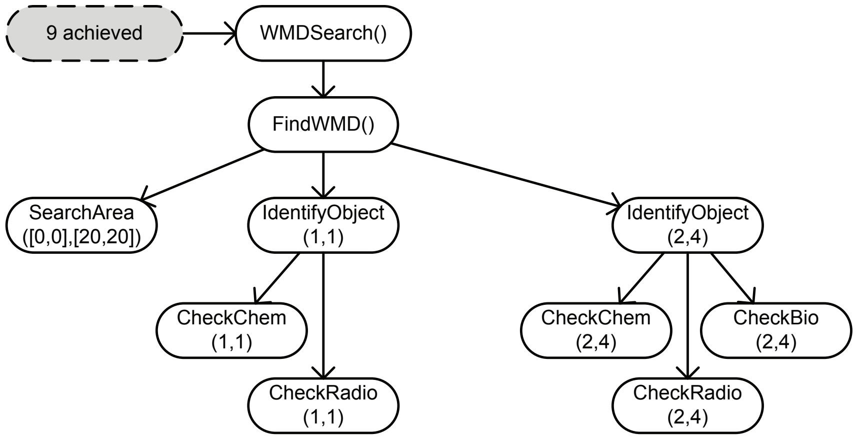

Figure 11. CheckBio(1,1) is achieved

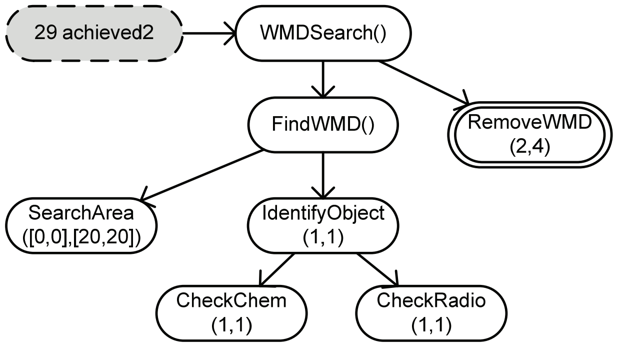

Figure 12. RemoveWMD goal created

There are several snapshots between Figure 12 and Figure 13. Two additional *RemoveWMD* goals were triggered and the *IdentifyObject(1,1)* goal was achieved. Although there are no *IdentifyObject* goals in G_I, the three *RemoveWMD* goals remain preceded since the *SearchArea([0,0],[20,20])* goal is still active and could potentially trigger other *IdentifyObject* goal. Once the SearchArea([0,0],[20,20]) goal is achieved, as shown in Figure 14, the *RemoveWMD* goals may be pursued. Once all the *RemoveWMD* goals are achieved, the system achieves its objective, *WMDSearch*.

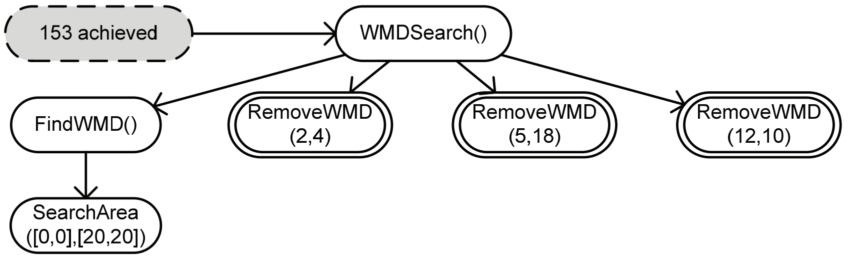

Figure 13. No more IdentifyObject goals

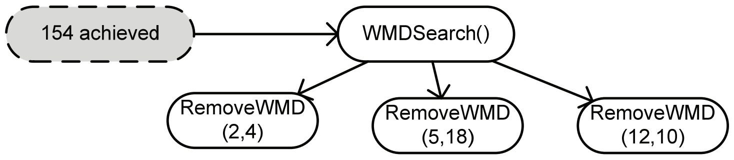

Figure 14. SearchArea Achieved

4 Use of GMoDS

GMoDS has been used in several ways, most notably as the requirements modeling for the Organization-based Multiagent Systems (O-MaSE) methodology [4] and as the runtime goal model for the Organization Model for Adaptive Complex Systems (OMACS) [2]. GMoDS has been used as part of O-

MaSE to model several multiagent and cooperative robotics applications. Many of these systems have used the runtime executability of GMoDS as part of various OMACS systems. Application areas that GMoDS has been used include adaptive information systems [2], medical decision supports systems [13], and multi-robotic search systems [1].

5 Conclusions and Future Work

Several frameworks have been developed for analyzing system goals. While each is adequate for static systems, none provides a continuous modeling/execution framework that ensures the goals identified actually drive the system during execution. GMoDS provides end-to-end modeling and execution and allows systems to adapt to changes in the environment and problem, a significant advantage for complex, adaptive systems. The GMoDS specification model includes the notions of goals, goal decomposition, events, precedence, and goal instantiation. The GMoDS instance model captures the dynamics of the system state while maintaining the structure of the specification model. The execution model, which has been used in several multiagent and cooperative robotic systems, implements these models in an efficient manner.

Our plans include extending GMoDS to handle soft goals, goal preferences, and goal metrics. *Soft goals* are goals for non-functional requirements and have been shown to be useful in requirements modeling. *Goal preferences* allow the designer to define which disjunctive goals are preferred, while *goal metrics* allow the goal model to be evaluated quantitatively at design time. Some prospective metrics include *goal model flexibility*, which measures how well a system using a specific model can adapt to failures; *goal criticality*, which ranks how essential each goal is to achieving the overall goal; and *goal occurrence*, which measures how often a specific goal appears in successful system runs. In addition to these enhancements, we are investigating adding mechanisms for requiring *simultaneous execution* constraints that will allow us to create *n* instances of a single goal that must be assigned and pursued simultaneously in order to support cooperative coalitions.

6 Acknowledgements

This research was performed as part of grants provided by the Air Force Office of Scientific Research FA9550-06-1-0058 and the National Science Foundation IIS-0347545.

7 References

[1] DeLoach, S.A. Organizational Model for Adaptive Complex Systems. in Virginia Dignum (ed.) *Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI Global: Hershey, PA.

[2] DeLoach, S.A., Oyenan, W.H., and Matson, E.T. A Capabilities Based Model for Artificial Organizations. *Journal of Autonomous Agents and Multiagent Systems*. 16(1): 13-56.

[3] DeLoach, S.A., Wood, M.F., and Sparkman, C.H.. 2001. Multiagent Systems Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 11(3): 231-258.

[4] Garcia-Ojeda, J.C., DeLoach, S.A., Robby, Oyenan, W.H., and Valenzuela, J. O-MaSE: A Customizable Approach to Developing Multiagent Development Processes. In Michael Luck (eds.), *Agent-Oriented Software Engineering VIII: The 8th International Workshop on Agent Oriented Software Engineering (AOSE 2007),* LNCS 4951, 1-15, Springer-Verlag: Berlin.

[5] Georgeff, M. P. and Lansky, A. L. 1987. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence* (AAAI-87), 677–682. AAAI Press.

[6] Huget, MP. 2004. Representing Goals in Multiagent Systems. In *Proceedings of the 4th International Symposium on Agent Theory to Agent Implementation*, 588-593. Austrian Society for Cybernetic Studies.

[7] Juan, T., Pearce, A., and Sterling, L. 2002. ROADMAP: Extending the Gaia Methodology for Complex Open Systems. In *Proceedings of the First International Joint Conference on Autonomous Agents And Multiagent Systems*, 3-10. New York: ACM.

[8] Miller, Matthew A. 2007. Goal Model for Dynamic Systems. Master's Thesis, Dept. of Computing and Information Sciences, Kansas State University.

[9] Morreale, V., Bonura, S., Francaviglia, G., Centineo, F., Cossentino, M., and Gaglio, S. 2006. Goal-Oriented Development of BDI Agents: The PRACTIONIST Approach. In *Proceedings of the IEEE/WIC/ACM international Conference on intelligent Agent Technology* (IAT), 66-72. IEEE Computer Society.

[10] Mylopoulos, J., Castro, J., and Kolp, M. 2000. Tropos: A Framework for Requirements-Driven Software Development. In J. Brinkkemper and A. Solvberg, editors, Information Systems Engineering: State of the Art and Research Themes. Berlin: Springer-Verlag.

[11] Odell, J., Parunak, H.V.D. & Bauer, B. 2000. Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop*.

[12] van Lamsweerde, A. and Letier, E. 2000. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Trans. on Software Engineering*. 26(10): 978-1005.

[13] Wilk, S., Michalowski, W., O'Sullivan, D., Farion, K., and Matwin, S. Engineering of a Clinical Decision Support Framework for the Point of Care Use. *American Medical Informatics Association (AMIA) Annual Symposium Proceedings 2008*.

[14] Yu, E. 1995. Modeling Strategic Relationships for Process Reengineering. PhD thesis, University of Toronto.