Proceedings of The National Conference
On Undergraduate Research (NCUR) 2015
Eastern Washington University, Cheney, WA
April 16-18, 2015

# An Architecture For Intentional Agents With Reactive Behavior

Alex Gain, Graham Haug
Department of Computer Science
Texas Tech University
2500 Broadway
Lubbock, Texas 79409 USA

Faculty Advisor: Yuanlin Zhang

## Abstract

This paper is centered on the design and implementation of intelligent agents. Recently, an architecture has been proposed which relies on the notion of intention[1]. In this work, intentions are related to agent beliefs, goals, and actions and are employed to solve a key problem at the center of intelligent behavior: selecting the next action for the agent to perform. However, this existing architecture results in agents which are driven only by their goals but cannot respond to domain triggers in a timely manner. For example, if an agent populates a room and the room is observed to be on fire, the agent's next action should be to leave the room regardless of its current goal and plan. The goal of our research is to design and implement an architecture for intentional agents that can react to environmental triggers regardless of the current goals and plans and allow for the agent to resume goal-driven behavior afterwards. Our design is based on well-established knowledge representation paradigms: Answer Set Prolog (ASP) and Action Language (AL). These languages have been very effective for describing possible agent actions and details of an agent's environment, both of which contribute to the agent's knowledge base. In order to expand the functionality of agents to allow for reactive behavior, we first modify Action Language (AL) to allow statements for triggered actions that describe how agents should respond to outside changes. We explicitly define the syntax and semantics for this newly modified AL. We extend the Theory of Intentions (TI) to allow these triggered actions to be treated as intended actions. We design and write ASP programs which will automatically implement any necessary reasoning tasks in the architecture. With these extensions, the architecture for intentional agents is expanded to allow agents featuring reactive behavior capability. Lastly, we discuss ideas for future improvements and expansions. Our research demonstrates that we can design intention-driven agents that can react to dynamic environments by utilizing Action Language and Answer Set Prolog.

**Keywords: Artificial Intelligence, Knowledge Representation, Declarative Programming**

## 1 Introduction

The Architecture for Intentional Agents[1] (AIA) is a means for the design and implementation of intelligent agents. Agents under the AIA architecture are intention-driven, creating plans and following them in order to achieve a goal in a dynamic environment. They are necessarily rational, that is they only believe what they are forced to believe by logical consequence of given constraints and rules. They are capable of making inferences about the past to explain unexpected observations.

For example, if an agent observes a glass of water on a table at one moment in time and then later observes it in a different location, it may update its history to include the action of someone moving it.

A missing functionality of agents under the AIA architecture is the ability to instinctively react to triggers in the environment and then resume intentional behavior. Consider the following example:

**Example** *[Building on Fire]*
Consider the case where intelligent agent Bob has a goal to meet his coworker John in their company building. Bob

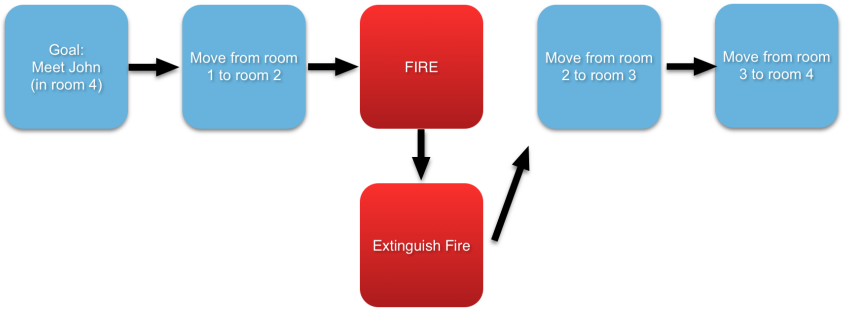


Figure 1: Illustration of *Building on Fire* example.

is initially in room 1 and John is initially in room 4. Bob, under the AIA architecture, is intention-driven and starts walking to room 4 to achieve his goal as quickly as possible. However, the building catches on fire. Because *fire* is a trigger, Bob puts his plan on hold and runs out of the building. Once he is safe and outside, he resumes his goal to meet John.

The existing AIA architecture does not allow for this type of behavior. To achieve this, Action Language[3] (AL) needed to be expanded, several definitions and formalisms involving notions of triggers needed to be introduced, and the previous AIA architecture needed to be modified.

## 2 Background

### 2.1 Answer Set Prolog (ASP)

ASP is a declarative language for knowledge representation. Roughly speaking, its statements are logical statements of the following forms: literals (facts), constraints, and if-then statements. Given a collection of ASP statements, ASP solvers can be used to produce an answer set — a set of beliefs which must logically follow from the given statements. Consider the following example:

**Example** *[ASP program]*
1. person(bob). *(This is a literal.)*
2. ¬dead(X) ← alive(X). *(This is an if-then statement and reads from right to left.)*
3. ← alive(X), dead(X). *(This is a constraint, saying that X cannot be both alive and dead at the same time.)*
4. alive(X) ← person(X), not dead(X). *(Here, "not" means "there is no reason to believe.")*

In English, the ASP program reads as:
1. "bob is a person."
2. "If X is alive then X is not dead."
3. "X cannot be both alive and dead at the same time."
4. "If X is a person and there is no reason to believe X is dead then X is alive."

There are some statements or knowledge which can be logically deduced from these rules. The collection of all of these statements is known as the answer set.

The answer set of this program is:
1. person(bob). *(Deduced from line 1 of the ASP program.)*
2. alive(bob). *(Deduced from lines 1 and 4 of the ASP program. The conditionals in line 4 are satisfied because we know bob is a person from line 1 and there is no reason to believe bob is dead)*
3. ¬dead(bob). *(Deduced from line 2 of this answer set and line 2 of the ASP program.)*

The answer set reads as:
1. “bob is a person.”
2. “bob is alive.”
3. “bob is not dead.”

The beliefs and conclusions of the agent are directly determined by the answer sets produced by the ASP programs of the agent. Thus, the agent is necessarily rational because it only believes what it is forced to believe by logical consequence.
Chapter 2 of reference 1 covers ASP more thoroughly than here.
Ultimately, the entire AIA architecture can be translated into ASP for implementation purposes.

## 2.2 Action Language (AL)

AL is a language which can be used to represent state transition systems[3]. Intuitively, AL can be thought of as representing a flowchart describing the state of the environment and the effects actions have on it.

Statements of AL contain *statics*, *fluents*, and *actions*. Fluents and statics can be thought of as properties of the state. Fluents can be changed through actions whereas statics cannot. A *domain literal* is a domain property $p$ or its negation $\neg p$ formed by a static or fluent. Literals formed by statics are called *static literals*, and literals formed by fluents are called *fluent literals*.
Consider the following example which will illustrate the type of statements allowed in AL:

**Example** *[AL: Chessboard]*
Consider a chessboard. The current position of the board and its pieces is the *current state* in AL. The starting position of the board would be the *initial state* in AL. Changes to this state can be described through *actions*. For example:

*move*(X, P) **causes** *at*(X, P)

Where *move* is an action and *at* is a fluent literal. This statement means moving piece X to position P will cause X to be at position P. The position of a piece is considered a fluent literal because it can be changed through actions. Contrarily, domain properties describing the layout of the board are considered static literals because they cannot be changed through actions.
Other statements of AL are *state constraints* and *executability conditions*. For example:

*alive*(X) **if** *at*(X, P)

This is a state constraint and means that piece X is alive if piece X is at some position on the board. Another example:

**impossible** *move*(X, P) **if** *captured*(X)

This is an executability condition and means that it is impossible to move piece X if piece X is captured.
Figure 1 illustrates this example.

The following is a formal definition of what was just covered:

**Definition 1 (Statements of AL)** *Language AL allows the following types of statements:*
1. *Causal Laws:*

$$a \textbf{ causes } l_{in} \textbf{ if } p_0, ..., p_m$$

2. *State Constraints:*

$$l \textbf{ if } p_0, ..., p_m$$

3. *Executability Conditions:*

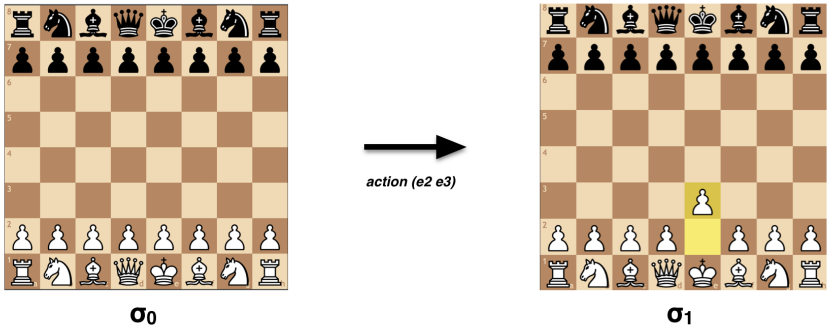


Figure 2: A chess example illustrating action language, where $\sigma_0$ and $\sigma_1$ are states and $action(e2e3)$ changes $\sigma_0$ to $\sigma_1$.

**impossible** $a_0, ..., a_k$ **if** $p_0, ..., p_m$

where $a$ is an action, $l$ is an arbitrary domain literal, $l_{in}$ is a literal formed by an inertial fluent, $p_0, ..., p_m$ are domain literals, $k > 0$, and $m \geq 1$. Moreover, no negation of a defined fluent can occur in the heads of state constraints.[1]

All AL statements have translations into ASP, seen in Chapter 2 of reference 1. Additionally a more formal and thorough review of AL can be found in Chapter 2 of reference 1.

## 2.3 The Architecture for Intentional Agents (AIA)

Agents under the AIA architecture are intention-driven, necessarily rational, and work to complete their goal as soon as possible. This section will give an overview of the AIA architecture. There are more formal and precise explanations of the AIA architecture[1,2]. For purposes of reading this paper, this concise section should provide enough of an intuitive understanding of AIA for understanding the rest of this paper.

The AIA control loop describes the behavior of an intentional agent under the AIA architecture and was originally the AAA control loop[4].

Observe the world and initialize history with observations;
1. interpret observations;
2. find an intended action $e$;
3. attempt to perform $e$ and update history with a record of the attempt;
4. observe the world, update history with observations, and go to step 1.

Figure 2: *AIA* Control Loop: The behavior of the intentional agent.

Multiple parts come together to produce this behavior. The AIA architecture can separated into these parts:

1. Theory of Intentions (TI)
2. Model of History
3. Description of Environment

For steps 1, 3, and 4 of Figure 1, Model of History and Description of Environment is needed. For step 2, *TI* is needed.

### *2.3.1 Theory of Intentions (TI)*

*TI* is a collection of Action Language *AL* statements. It is a representation of properties of intention. *TI* is necessary for the agent's capability of selecting a goal, creating and selecting plans, and executing plans. Intuitively, it can be thought of as comprising the mental state and mental rules of the agent.
As an example, *select* and *abandon* are for selecting and abandoning goals.

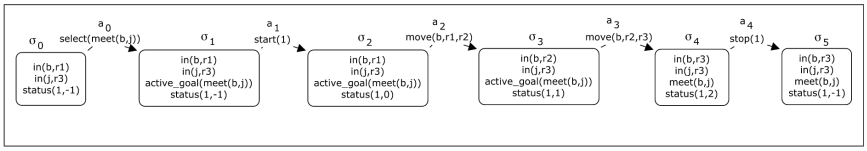


Figure 3: Example of an *AL* transition diagram for the agent's mental and physical states.

*select*(G) **causes** *active_goal*(G).
*abandon*(G) **causes** *active_goal*(*G*).
where G is a goal.

As another example, *status* is for knowing if an activity (plan to achieve the goal) is active and what step it is on. *start* will start an activity by making its status active. Similarly, *stop* will stop an activity by making its status inactive. For example status(bake_bread, 0) means activity (i.e. plan) bake_bread is active, status(bake_bread, 2) means activity bake_bread is active and on step 2, and status(bake_bread, -1) means activity bake_bread is inactive. The following are the *AL* statements for *start* and *stop*:

*start*(M) **causes** *status*(M, 0).
*stop*(M) **causes** *status*(M, 1).

Additional statements specify how the status of the current plan, i.e. what step the plan is currently on, is incremented[2].

The most important aspects of *TI* are keeping track of the status of plans and finding the next intended action of the agent. As an example, here is one of the ASP statements for finding the intended action of the agent:

intended action(stop(M), I) ← current_step(I), interpretation(N, I), category_3(M, I), futile(M, I).

There is more background needed than what is covered here to fully understand this statement. For instance, "category_3" refers to category of history which will be covered later. However, intuitively this statement says that if the agent is in the middle of executing a plan, but the plan is not expected to achieve the goal, then the next action of the agent should be to stop its current plan. There are several statements such as this in *TI* which specify what the agent's intended action should be in various situations.
There are many more AL and ASP statements for *TI* than mentioned here[2].
Figure 3 shows an *AL* transition diagram that shows the changes in the agent's mental and physical states[1].

### *2.3.2 Model of History*

A model of history is needed for the agent to keep a coherent and rational view of the world, the world's current state, and what actions and observations necessitated the current state.

The model of history consists of the following statements: *hpd* and *obs*, where *hpd* corresponds to what actions have occurred and *obs* corresponds to what fluents were true. For example, a history could be:

1. obs(in(john, room3), 1).
2. hpd(moves(room4), 2).
3. obs(in(john, room4), 3).

In English, this means:
1. "John was observed to be in room 3 at step 1."
2. "It happened that John moved to room4 at step 2."
3. "John was observed to be in room 4 at step 3."

### *2.3.3 Exogenous Actions and Updating History*

Actions occurring in that environment that are not performed by the agent are called *exogenous actions*. Sometimes, an exogenous action may have occurred without the agent's knowledge or observation — an unobserved exogenous action. Unobserved exogenous actions may cause changes in the state that are unexpected. For example, consider the following history:

1. obs(on(water1, table1), 1).
2. obs(on(water1, table2), 2).

In English, this means:
1. "Water1 is observed to be on table1 at step 1."
2. "Water1 is observed to be on table2 at step 2."

Sometime during step 1, the water must have been moved from table1 to table2. Thus, the agent will update its history to reflect this:

1. obs(on(water1, table1), 1).
2. hpd(moved(water1, table2), 1).
3. obs(on(water1, table2), 2).

This update allows the agent to maintain a coherent model of the world and allows the agent to make inferences on unexpected observations.

### *2.3.4 Description of Environment*

In addition to $TI$ and rules of history, the agent must have rules describing the environment it is in. These rules may be provided in the form of AL statements and corresponding ASP translations. For instance, the rules found in **Example** *[AL: Chessboard]* would be a good description of the environment if that is the type of environment the agent is in.

## 3 Expanding Action Language (AL)

In order to expand the functionality of the agent to allow for reactive behavior, we first expand Action Language (AL) to allow statements for triggered actions that describe how agents should respond to outside changes. Formalisms and definitions are introduced to define the syntax and semantics of these expansions.

As seen in **Definition 1** under Section 2.2 Action Language (*AL*), there are three types of statements allowed in *AL*. We add a fourth statement to allow for triggers:

*Trigger statements:*

$$l \textbf{ triggers } a.$$

As an example, a trigger statement could be

$$building\_on\_fire \textbf{ triggers } run\_outside(\text{bob})$$

which means $building\_on\_fire$ triggers bob to run outside the building.
To define the semantics of triggers and triggered actions, the following definitions and formalisms are introduced:

**Definition 2** [*Triggered actions*]
Let $\sigma_0$ be a state. For every trigger statement $l$ **triggers** $a$, if $l$ exists in $\sigma_0$ and a does not lead to an inconsistency in $\sigma_1$ then $a$ is a *triggered action* of $\sigma_0$.
**Definition 3** [*Triggered agent actions*]
If $b$ is a physical agent action and $b$ is a triggered action, then $b$ is a triggered agent action.

The definition of transition[1] is modified:

**Definition 4** [*Transition*]
Let $a$ be a non-empty collection of actions and $\sigma_0$ and $\sigma_1$ be states of transition diagram $T(SD)$ defined by system description $SD$.
A state-action-state triple $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition of $T(SD)$ if every triggered action of $\sigma_0$ belongs to $a$ and $\Pi(SD, \sigma_0, a)$ has an answer set $A$ such that $\sigma_1 = \{l : h(l,1) \in A\}$.

With respect to the order in which the previous statements were introduced, the corresponding ASP translations are as follows:

1. triggered_action(A, I) ← holds(F, I).
2. occurs(A, I) ← triggered_action(A, I), not ¬occurs(A, I).
3. triggered_agent_action(A,I) ← agent_action(A), triggered_action(A, I).

Which are read as:

1. "If fluent F holds at step I then A is a triggered action at step I."
2. "If A is a triggered action at step I and there is no reason to believe A does not occur at step I, then A occurs at step I."
3. "If A is an agent action and A is a triggered action at step I then A is a triggered agent action at step I."

The function of Line 1 is to establish triggers and mark when there is a triggered action.

Line 2 is needed to ensure that triggered actions occur. The "not ¬occurs(A, I)" in line 2 is needed because there are situations where a trigger may fail. For instance, a building on fire may trigger the agent to run outside the building, but if the agent is unconscious during this time then the agent cannot run outside the building. The answer set would be inconsistent rendering the agent dysfunctional. The "not occurs(A, I)" accounts for situations such as these and keeps the answer set consistent.

Line 3 marks when a triggered agent action occurs and is needed for expanding *TI*, which will be explained in the next section.

## 4 Expanding *TI*

Under the AIA architecture, it is impossible for an agent to perform an action that is not intended. For this reason, *TI* must be modified so that if an agent action is triggered then that triggered action must be the agent's next intended action.

In *TI*, there are rules for intended actions of the agent. The intended action of the agent is based on the recorded history. For instance, if there are no active goals then the history is of category 1 and the agent's intended action is to wait. If there is an active goal and an active plan that is predicted to succeed in achieving the goal then the history is of category 3 and the agent's intended action is the next step of the plan.

The following definitions, AL statements, and ASP rules are part of TI and involve categories of history and intended actions[1,2].

**Definition 5.** [*Categories of histories*]
Let $cm_n$ be the current mental state of history $\Gamma_n$.

*category 1* - there are no activities or goals that are active in $cm_n$;
*category 2* - there is an activity $m$ such that $m$ is top-level and active in $cm_n$ but its goal $g$ is no longer active in $cm_n$ (i.e the goal is either achieved or abandoned);
*category 3* - there is an activity $m$ such that $m$ and its goal $g$ are both top-level and active and $a$ is the next action of $m$ in $cm_n$;
*category 4* - there is a goal $g$ that is active in $cm_n$ but no activity with goal $g$ is active in $cm_n$;

We add a new category, category 5, for when there is a triggered agent action:

*category 5* - there is a triggered action $b$ in state $\sigma_n$ such that $\sigma_n$ is a possible current state of history and $b$ is a physical agent action.

We define the intended action for category 5:

**Definition 6.** [*Intended triggered action of a history of category 5*]
Triggered agent action $b$ is an intended action of history $\Gamma_n$ if $b$ is a triggered agent action of current state of $\Gamma_n$.

We provide ASP translations for these additions:

category_5(I) :– current_step(I), interpretation(N, I), triggered_agent_action(B, I).
intended action(B, I) :— current step(I), interpretation(N, I), category 5(I), triggered_agent_action(B, I).

We amend the other categories so that category 5 takes precedence over the other categories:

*category 1* - there are no activities or goals that are active in $cm_n$ and history is not of category 5;
*category 2* - there is an activity $m$ such that $m$ is top-level and active in $cm_n$ but its goal $g$ is no longer active in $cm_n$ (i.e the goal is either achieved or abandoned) and history is not of category 5;
*category 3* - there is an activity $m$ such that $m$ and its goal $g$ are both top-level and active and $a$ is the next action of $m$ in $cm_n$ and history is not of category 5;
*category 4* - there is a goal $g$ that is active in $cm_n$ but no activity with goal $g$ is active in $cm_n$ and history is not of category 5;

We modify the ASP translation statements of categories of history:

category 1(I) :— current step(I), interpretation(N, I), not active goal or activity(I), not category_5(I).
category 2(M, I) :— current step(I), interpretation(N, I), ¬h(minor(M), I), h(active(M), I),
goal(M, G), ¬active goal(G), not category_5(I).
category 3(M, I) :— current step(I), interpretation(N, I), ¬ h(minor(M), I), h(in progress(M), I),
not category_5(I).
category 4 history(G, I) :— current step(I), interpretation(N, I), ¬h(minor(G), I), h(active goal(G), I),
¬h(in progress(G), I), not category_5(I).

Finally, by modifying an ASP implementation of the *AIA* architecture and using a simple context for the agent to operate in, the agent achieved reactionary behavior as expected.

## 5 Implementation

Using the ASP modifications introduced, we were able to produce an ASP implementation of an intelligent agent under the AIA architecture which responded to domain triggers and returned to its intention-driven behavior afterwards. We placed the agent in several scenarios and evaluated its performance.

### 5.1 Scenario 5.1

Consider the following context that the agent was placed in: The location of the agent *bob* is in a building with four rooms $r1$, $r2$, $r3$, and $r4$ that are linearly connected. A coworker *john* is situated in $r4$ and *bob* is situated in $r1$.

The goal of *bob* is to meet with *john*. Thus, once the goal is selected, *bob* will immediately start moving from $r1$ to $r4$. However, if there is a fire in the building then this will trigger *bob* to extinguish the fire. This is the situation the agent was placed in.

During steps 0 and 1, the goal was selected and a plan was started to achieve the goal. During step 2, *bob* moved from $r1$ to $r2$. At step 3, there was a fire in the building. Instead of moving from $r2$ to $r3$ during this step, *bob* reacted

to the trigger as desired and extinguished the fire. During step 4, *bob* resumed working to achieve the goal and moved from $r2$ to $r3$.

### 5.2 Scenario 5.2

In this scenario, we wanted to test the case where the agent repeatedly performs a triggered action in an unintelligent way. The phone was set to ring at every step. So, at every step bob answered his phone. To change this behavior, we added some statements that prohibited bob from answering the phone more than three steps in a row. This successfully caused the unintelligent behavior to stop and for bob to achieve his goal.

### 5.3 Scenario 5.3

Scenario 3 was the same as scenario 1 except another trigger was added: Bob's phone ringing triggers bob to answer his phone, and it was set so that bob's phone rings at step 3. Additionally, a fire was set to occur at step 3 as well. At step 3, as in scenario 1, bob put his goal temporarily on hold and attempted to extinguish the fire. At the same time, bob answered his phone. This is not necessarily behavior that we consider intelligent. We want bob to ignore his phone and extinguish the fire. So, additional statements were added that assigned priorities to the two triggers. The fire trigger was assigned higher priority. With these additional statements added, bob extinguished the fire at step 3 and ignored his phone.

These different scenarios were tested for a couple reasons. Firstly, we wanted to make sure the basic functionality of the trigger worked. Secondly, we wanted to examine some problem scenarios where triggers might not work and what we can do to fix them. Given the scenarios considered and the results obtained, it appears that triggers are flexible enough to accommodate most issues that may arise from them.

Regarding the programming language of the implementation, an old version of sparc[5], which is a declarative programming language, was used. Included below is a screenshot of the command-line output displaying a subset of the Answer Set described containing the actions just described.

```
program translated
Alexs-MacBook-Air-2:impnew doctorjuice$  ./dlv.bin -nofinitecheck -pfilter="occurs" new
DLV [build BEN/Dec 17 2012   gcc 4.2.1 (Apple Inc. build 5666) (dot 3)]

Best model: {occurs(select(meet(b,j)),0), occurs(extinguish_fire(b),3), occurs(move(b,1,2),2),
occurs(move(b,2,3),4), occurs(move(b,3,4),5), occurs(stop(1),6), occurs(start(1),1)}
Cost ([Weight:Level]): <[3:1]>
```

Figure 2: A subset of the implementation answer-set containing the actions of the agent.

The ASP program written for the implementation and accompanying files can be found on github by searching AIAImplementation.

## 6 Discussion and Possible Expansions

The AL expansions and TI modifications introduced successfully expanded the functionality of agents under the AIA architecture to allow for reactive behavior.

Although our implementations produced reactionary behavior as expected, it is possible that a more complex situation will yield different results. Multiple types of agent triggers may lead to inconsistent answer sets or the agent perpetually trying to perform a triggered action but being unable to do so. Thus, depending on the context of the agent, the number of types of triggered actions the agent is capable of performing may be limited. At worst, the agent may be limited to a single type of agent action trigger. A proposed quick fix for this would be to limit the number of times in a row the agent can attempt to execute a specific agent action trigger. This was done for a specific case in scenario 2 with success.

Additionally, if an agent runs into problems trying to execute more than one trigger at once, integer numbers can be assigned to agent triggers to denote the priority of those triggers. For example, consider triggered_agent_action(a, I, p) and triggered_agent_action(b, I, k), where $a$ and $b$ are different actions, $I$ is the current step, and $p$ and $k$ are integer values representing priority. If $p > k$ then $a$ should be the intended action of the agent and if $k > p$ then $b$ should be

the intended action of the agent. In other words, assigning priority to agent triggers can ensure that specific triggers are not executed simultaneously. A specific case of this was implemented in scenario 3 and worked for that specific case.

Regardless, as long as the designer of the specific AIA agent is cognizant and chooses what triggers to give the agent in a thoughtful manner, these issues should not arise, as demonstrated in our implementations.

Another possible expansion of the AIA architecture would be to allow for the selection of more than one goal at a time. Consider the following scenario: Agent bob needs to buy eggs and milk at the grocery store. He is currently at his house. There are of couple ways bob can go about achieving these goals. He could go the grocery store, buy eggs, and return to his home. Afterwards, he could go to the grocery store, buy milk, and return to his home. However, the most efficient way to achieve these two goals would be for bob to go to the grocery store, buy both eggs and milk, and return home.

The current AIA architecture only allows for one goal selection at a time. A valuable expansion would be to allow for multiple goals at a time and for the agent to behave in the manner discussed.

With regard to the scope of this paper however, the agent produced reactionary behavior as desired, which the results of the implementations show. The definitions, formalisms, and ASP translations introduced provide a significant expansion to the original AIA architecture and allow agents under the AIA architecture to be successful in many more contexts.

## 7 Related Works

As mentioned before, this work is an extension of Blount's *AIA* architecture[1], which is itself based on several earlier works. In the earlier *AAA* architecture[4], *AL*[6] was used to describe the environment the agent was in and record the history of the agent. The *AAA* architecture had an agent working towards a goal in a changing environment, with some main differences between *AAA* and *AIA* being more dynamic planning and a more organized and cohesive control loop. Furthermore, because the notion of intention is formally defined in *AIA*, statements about the overall behavior of an agent under the *AIA* architecture can be formally reasoned about or analyzed. These formal statements of intention were previously well-defined by Baral and Gelfond[7].

Since the *AIA* architecture makes formal statements about intention and behavior, it falls under the category of a BDI model[8,9].

Another work *AIA* relies on are consistency-restoring rules[10], which are used in *TI* in order to be able to replan and explain unexpected observations. Since this paper is about extending *AIA*, all the conceptual frameworks outlined here that *AIA* falls under, also encompass the extended *AIA* architecture in this paper.

## 8 Acknowledgements

The authors wish to express their appreciation for Michael Gelfond in discussing and guiding the direction of the project, Yuanlin Zhang for his thorough and careful work in instructing and teaching the research process, and Evgenii Balai for his technical expertise and help with the implementation.

## 9 References

1. Blount, J., "An Architecture For Intentional Agents," Ph.D. Dissertation, Texas Tech University (December 2013).

2. Blount J., M. Gelfond, and M. Balduccini, "Towards a Theory of Intentional Agents," AAAI Stanford Spring Symposium on Knowledge Representation and Reasoning in Robotics (2014).

3. Gelfond, M. and V. Lifschitz, "Action Languages in Linkping Electronic Articles in Computer and Information Science," Vol. 3 (1998), URL http://www.ep.liu.se/ea/cis/1998/016/.

4. Marcello, B. and M. Gelfond, "The AAA Architecture: An Overview," In AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents (AITA08) (Mar 2008).

5. Balai, E., M. Gelfond, and Y. Zhang, "Towards Answer Set Programming with Sorts," In Proceedings of LPNMR-2013 (2013).

6. Baral, C., and M. Gelfond, "Reasoning Agents In Dynamic Domains," In Workshop on Logic-Based Artificial Intelligence (2000).

7. Baral, C., and M. Gelfond, "Reasoning about Intended Actions," In Proceedings of AAAI05, 689694 (2005).

8. Rao, A. S., and M. P. Georgeff, "Modeling rational agents within a BDI-architecture," In Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning, 473484 (1991).

9. Wooldridge, M., "Reasoning about Rational Agents," The MIT Press (2000).

10. Balduccini, M., and M. Gelfond, "Logic Programs with Consistency-Restoring Rules," In Doherty, P.; McCarthy, J.; and Williams, M.-A., eds., International Symposium on Logical Formalization of Commonsense Reasoning, AAAI Spring Symposium Series, 918 (2003).