Action Language Hybrid AL

Alex Brik[1] and Jeffrey Remmel[2]

[1] Google Inc
[2] Department of Mathematics, UC San Diego

Abstract. This paper introduces an extension of the action language $\mathcal{AL}$ to Hybrid $\mathcal{AL}$. A program in Hybrid $\mathcal{AL}$ specifies both a transition diagram and associated computations for observing fluents and executing actions. The semantics of $\mathcal{AL}$ is defined in terms of Answer Set Programming (ASP). Similarly, the semantics of Hybrid $\mathcal{AL}$ is defined using Hybrid ASP which is an extension of ASP that allows rules to control sequential execution of arbitrary algorithms.

Constructing a mathematical model of an agent and its environment based on the theory of action languages has been studied and has applications to planning and diagnostic problems, see [10] for an overview. In the realm of diagnostic problems, the goal is to find explanations of unexpected observations. We are interested in solving diagnostic problems such as those that arise diagnosing malfunctions of a large distributed software system, as described in [14].

The approach to solving a diagnostic problems described in [1] is based on the idea of using a mathematical model of the agent's domain, created using a description in the action language $\mathcal{AL}$ [2] to find explanations for unexpected observations. Central to this approach is the notion of the **agent loop** [10] which we modify to underline the relevance to the diagnostic problem.
1. Observe the world, check that observations are consistent with expectations, and update the knowledge base.
2. Select an appropriate goal G.
3. Explain unexpected observations and search for a plan (a sequence of actions) to achieve G.
4. Execute an initial part of the plan, update the knowledge base, go back to step 1.

The description and the facts from the knowledge base are translated into a logic program in a language of answer set programming (ASP) [11]. An ASP solver is then used to find stable models of the program, which are descriptions of possible trajectories of the underlying domain. These can be used to carry out steps 1 and 3 of the agent loop.

The two assumptions for the applicability of the agent loop are: (1) the agent is capable of making correct observations, performing actions, and recording these observations and actions (not defeasible), and (2) normally the agent is capable of observing all relevant exogenous actions occurring in its environment (defeasible). Hybrid $\mathcal{AL}$ is introduced to help solve a diagnostic problem where both (1) and (2) are defeasible, and where to decrease the size of the search space the following are assumed: (A1) the agent is normally capable of determining a small set of possible actions occurring in its environment by computationally simulating its environment for a number of steps starting from a known state in the past, or by performing other relevant external computations (here, by a small set we mean a set that can practically be represented by the enumeration), (A2) the agent has a description of at least one past state, and that description is sufficient to satisfy the assumption A1.

Under these assumptions, the agent may need to perform sequential computations (where the choice of the computations at step j may depend on the output of computations at step $j-1$) in order to determine sets of possible actions and states of the domain. Our hypothesis is that under these assumptions, computational efficiency can be improved if ASP-like processing and the external computations are merged. Such a merging can reduce the number of possible actions and states that need to determine the next action.

One way to address such issues is to extend $\mathcal{AL}$ to a richer action language that provides mechanism for performing computations and passing input and output parameters between the computation steps. Hybrid $\mathcal{AL}$, introduced in this paper is one such extension. While descriptions in $\mathcal{AL}$ are translated into ASP, the descriptions in Hybrid $\mathcal{AL}$ are translated into Hybrid ASP (H-ASP) which is an extension of ASP, introduced by the authors in [3] that allows rules to control sequential execution of arbitrary algorithms. This functionality makes H-ASP well suited for solving a diagnostic problem under our assumptions.

The outline of this paper is as follows. In section 1, we will discuss an example to motivate our need to extend $\mathcal{AL}$ to Hybrid $\mathcal{AL}$, and we will briefly describe $\mathcal{AL}$ and H-ASP. In section 2 we will introduce Hybrid $\mathcal{AL}$. In section 3 we will revisit the example from section 1 and show how it can be described in Hybrid $\mathcal{AL}$. Section 4 contains discussion of related work and conclusions.

1 Motivation for Hybrid $\mathcal{AL}$ and Preliminaries

To motivate the introduction of Hybrid $\mathcal{AL}$, we will consider an example of a hypothetical video processing system (figure 1). The system selects a video from a video library. The choice depends on the initial state and time. It then checks the quality of the video. If the check fails, then a system is said to malfunction, which is unexpected.

We are interested in creating a diagnostic agent of this system. We will assume that the agent is only able to determine a subset of the videos $v_1, ..., v_k$ dependent on the initial state, one of which was chosen by the system. The agent will then investigate k possible trajectories of the system - one for each possible video. For each such trajectory, the agent will check quality of the video and determine whether it could have caused unexpected behavior.

Since the subset of the videos selected by the agent is time dependent, it may not be practical to pre-compute the values of properties of the videos. Instead, if the system malfunctions, then the agent will access the video library, select a

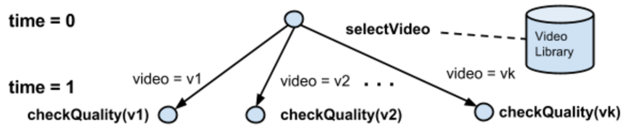

Fig. 1. Video processing system

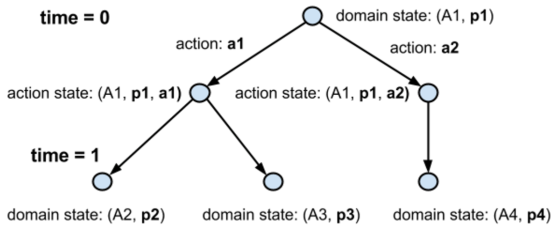
time = 0
domain state: (A1, **p1**)
action: **a1**
action: **a2**
action state: (A1, **p1**, **a1**)
action state: (A1, **p1**, **a2**)
time = 1
domain state: (A2, **p2**)
domain state: (A3, **p3**)
domain state: (A4, **p4**)

Fig. 2. Hybrid transition diagram

set of videos and run the quality check algorithm on each of the selected videos to explain the unexpected behavior of the system.

A key concept related to action languages is that of a transition diagram, which is a labeled directed graph where vertices are states of a dynamic domain, and edges are labeled with subsets of actions. In Hybrid $\mathcal{AL}$, one considers **hybrid transition diagrams**, which are directed graphs with two types of vertices: action states and domain states. A **domain state** is a pair $(A, \mathbf{p})$ where A is a set of propositional atoms and $\mathbf{p}$ is a vector of sequences of 0s and 1s. We can think of A as a set of values of the properties of a system, and $\mathbf{p}$ as the description of the parameters used by external computations. An **action state** is a tuple $(A, \mathbf{p}, a)$ where A and $\mathbf{p}$ are as in the domain state, and a is a set of actions. An out edge from a domain state must have an action state as its destination. An out edge from an action state must have a domain state as its destination. Moreover, if $(A, \mathbf{p})$ is a domain state that has an out-edge to an action state $(B, \mathbf{r}, a)$, then $A = B$ and $\mathbf{p} = \mathbf{r}$. There is a simple bijection between the set of transition diagrams and the set of hybrid transition diagrams.

An example of a hybrid transition diagram is in Figure 2. Two actions $a1$ and $a2$ can be performed in the state $(A1, \mathbf{p}1)$. Thus, the action states are $(A1, \mathbf{p}1, a1)$ and $(A1, \mathbf{p}1, a2)$. The consequents of applying action $a1$ at the state $(A1, \mathbf{p}1)$ are two domain states $(A2, \mathbf{p2})$ and $(A3, \mathbf{p3})$, for action $a2$ at $(A1, \mathbf{p}1)$ it is the domain state ($A4$, **p4**).

We will now briefly review action language $\mathcal{AL}$. Our review is based on Chapter 8 of [10] where one can find more details. $\mathcal{AL}$ has three special sorts: **statics, fluents,** and **actions**. The fluents are partitioned into two sorts: **inertial** and **defined**. Statics and fluents are referred to as **domain properties**. Intuitively, statics are properties of the system that don't change with time. Inertial fluents are properties that are subject to the law of inertia. Their values can be directly influenced by actions, and in the absence of such a change the values remain unchanged. Defined fluents are properties defined in terms of other fluents and cannot be directly influenced by actions. A **domain literal** is a domain property p or its negation $\neg p$. If a domain literal l is formed by a fluent, it is referred to as **fluent literal**; otherwise it is a **static literal**. A set S of domain literals is called **complete** if for any domain property p either p or $\neg p$ is in S. S is called **consistent** if there is no p such that $p \in S$ and $\neg p \in S$.

$\mathcal{AL}$ allows the following types of statements:
1. **Causal Laws**: a causes l_{in} if $p_0, ..., p_m$,
2. **State constraints**: l if $p_0, ..., p_m$, and
3. **Executability conditions**: 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, $k \geq 0$ and $m \geq -1$.

A system description $\mathcal{SD}$ in $\mathcal{AL}$ specifies a transition diagram. Intuitively, state constraints specify sets of allowed states. Given a state S, the set of executability conditions specifies concurrent actions executable at S, i.e. sets of actions that can decorate out edges of S. Given a state S and a set of actions A, causal laws together with state constraints determine the set of possible consequent states that result from executing A at S, i.e. neighbor states connected to S by out edges decorated by A.

Formally, a complete and consistent set σ of domain literals is a state of a transition diagram defined by $\mathcal{SD}$ if σ is the unique answer set of program $\Pi_c(\mathcal{SD}) \cup \sigma_{nd}$, where σ_{nd} is the collection of all domain literals of σ formed by inertial fluents and statics (the definition of $\Pi_c(\mathcal{SD})$ is omitted for brevity).

A system description $\mathcal{SD}$ of $\mathcal{AL}$ is called **well founded** if for any complete and consistent set of fluent literals σ satisfying the state constraints of $\mathcal{SD}$, the program $\Pi_c(\mathcal{SD}) \cup \sigma_{nd}$ has at most one answer set. Sufficient conditions for well-foundedness are expressed in terms of the **fluent dependency graph**, which is a directed graph such that its vertices are arbitrary domain literals and where it has an edge: **(a)** from l to l' if l is formed not by a defined fluent and $\mathcal{SD}$ contains a state constraint with the head l and the body contains l', **(b)** from f to l' if f is a defined fluent and $\mathcal{SD}$ contains a state constraint with head f and body containing l' and not containing f, **(c)** from $\neg f$ to f for every defined fluent f. A fluent dependency graph is said to be **weakly acyclic** if it does not contain any paths from defined fluents to their negations.

Proposition (Proposition 1 in [9]) If a system description $\mathcal{SD}$ of $\mathcal{AL}$ is weakly acyclic, then $\mathcal{SD}$ is well-founded.

A transition $\langle \sigma_0, a, \sigma_1 \rangle$ is described in terms of a program $\Pi(\mathcal{SD}, \sigma_0, a)$ (the definition of $\Pi(\mathcal{SD}, \sigma_0, a)$ is omitted for brevity). A state-action-state triple $\langle \sigma_0, a, \sigma_1 \rangle$ is a **transition** of $\mathcal{T}(\mathcal{SD})$ iff $\Pi(\mathcal{SD}, \sigma_0, a)$ has an answer set A such that $\sigma_1 = \{l : h(l, 1) \in A\}$.

We now give a brief overview of H-ASP restricted to the relevant rules. A H-ASP program P has an underlying parameter space S and a set of atoms At. Elements of S, called *generalized positions*, are of the form $\mathbf{p} = (t, x_1, \ldots, x_m)$ where t is time and x_i are parameter values. We let $t(\mathbf{p})$ denote t and $x_i(\mathbf{p})$ denote x_i for $i = 1, \ldots, m$. The universe of P is $At \times S$. A pair $(Z, \mathbf{p})$ where $Z \subseteq At$ and $\mathbf{p} \in S$ will be referred to as a *hybrid state*. For $M \subseteq At \times S$, we write $\mathbb{GP}(M) = \{\mathbf{p} \in S : (\exists a \in At)((a, \mathbf{p}) \in M)\}$, $W_M(\mathbf{p}) = \{a \in At : \ (a, \mathbf{p}) \in M\}$, and $(Z, \mathbf{p}) \in M$ if $\mathbf{p} \in \mathbb{GP}(M)$ and $W_M(\mathbf{p}) = Z$. A *block* B is an object of the form $B = a_1, \ldots, a_n, \textit{not } b_1, \ \ldots, \textit{not } b_m$ where $a_1, \ldots, a_n, b_1, \ldots, b_m \in At$. We let $B^- = \textit{not } b_1, \ \ldots, \ \textit{not } b_m$, and $B^+ = a_1, \ldots, a_n$. We write $M \models (B, \mathbf{p})$, if $(a_i, \ \mathbf{p}) \in M$ for $i = 1, \ldots, n$ and $(b_j, \ \mathbf{p}) \notin M$ for $j = 1, \ \ldots, \ m$.

Advancing rules are of the form: $a \leftarrow B : A, O$. Here B is a block, $O \subseteq S$, for all $\mathbf{p} \in O$ $A(\mathbf{p}) \subseteq S$, and for all $\mathbf{q} \in A(\mathbf{p})$, $t(\mathbf{q}) > t(\mathbf{p})$. A represents a partial function $S \to 2^S$, and is called an *advancing algorithm*. The idea is that if $\mathbf{p} \in O$ and B is satisfied at $\mathbf{p}$, then A can be applied to $\mathbf{p}$ to produce a set of generalized positions O' such that if $\mathbf{q} \in O'$, then $t(\mathbf{q}) > t(\mathbf{p})$ and $(a, \ \mathbf{q})$ holds.

Stationary-i rules are of the form: $a \leftarrow B_1; B_i : H, O$ (where for $i = 1$ we mean $a \leftarrow B_1 : H, O$). Here B_i are blocks and H is a Boolean algorithm defined on O. The idea is that if $(\mathbf{p}_1, \ \mathbf{p}_i) \in O$ (where for $i = 1$ we mean $\mathbf{p}_1 \in O$), B_k is satisfied at $\mathbf{p}_k$ for $k = 1, i$, and $H(\mathbf{p}_1, \ \mathbf{p}_i)$ is true (where for $i = 1$ we mean $H(\mathbf{p}_1)$), then $(a, \ \mathbf{p}_i)$ holds.

A *H-ASP Horn program* P is a H-ASP program which does not contain any negated atoms in At. For $I \in S$, the one-step provability operator $T_{P,I}(M)$ consists of M together with the set of all $(a, J) \in At \times S$ such that **(1)** there exists a stationary-i rule $a \leftarrow B_1; B_i : H, O$ such that $(\mathbf{p}_1, \ \mathbf{p}_i) \in O \cap (\mathbb{GP}(M) \cup \{I\})^i$, $M \models (B_k, \ \mathbf{p}_k)$ for $k = 1, i$, and $H(\mathbf{p}_1, \mathbf{p}_i) = 1$, and $(a, \ J) = (a, \ \mathbf{p}_i)$ or **(2)** there exists an advancing rule $a \leftarrow B : A, O.$ such that $\mathbf{p} \in O \cap (\mathbb{GP}(M) \cup \{I\})$ such that $J \in A(\mathbf{p})$ and $M \models (B, \ \mathbf{p})$.

An advancing rule is *inconsistent* with (M, I) if for all $\mathbf{p} \in O \cap (\mathbb{GP}(M) \cup \{I\})$ either $M \not\models (B, \ \mathbf{p})$ or $A(\mathbf{p}) \cap \mathbb{GP}(M) = \emptyset$. A stationary-$i$ rule is *inconsistent* with (M, I) if for all $(\mathbf{p}_1, \ \mathbf{p}_i) \in O \cap (\mathbb{GP}(M) \cup \{I\})^i$ there is a k such that $M_k \not\models (B_k, \ \mathbf{p}_k)$ or $H(\mathbf{p}_1, \ \mathbf{p}_i) = 0$.

We form the Gelfond-Lifschitz reduct of P over M and I, $P^{M,I}$ as follows. **(1)** Eliminate all rules that are inconsistent with (M, I). **(2)** If the advancing rule is not eliminated by (1), then replace it by $a \leftarrow B^+ : A^+, O^+$ where O^+ is the set of all $\mathbf{p}$ in $O \cap (\mathbb{GP}(M) \cup \{I\})$ such that $M \models (B^-, \ \mathbf{p})$ and $A(\mathbf{p}) \cap \mathbb{GP}(M) \neq \emptyset$, and $A^+(\mathbf{p}) = A(\mathbf{p}) \cap \mathbb{GP}(M)$. **(3)** If the stationary-i rule is not eliminated by (1), then replace it by $a \leftarrow B_1^+; \ B_i^+ : H|_{O^+}, O^+$ where O^+ is the set of all $(\mathbf{p}_1, \ \mathbf{p}_i)$ in $O \cap (\mathbb{GP}(M) \cup \{I\})^i$ such that $M \models (B_k^-, \mathbf{p}_k)$ for $k = 1, i$, and $H(\mathbf{p}_1, \mathbf{p}_i) = 1$. Then M is a *stable model of P with initial condition I* if $\bigcup\limits_{k=0}^{\infty} T^k_{P^{M,I},I}(\emptyset) = M$.

We will now introduce additional definitions which will be used later on in this paper. We say that an advancing algorithm A lets a parameter y be *free* if the domain of y is Y and for all generalized positions $\mathbf{p}$ and $\mathbf{q}$ and all $y' \in Y$, whenever $\mathbf{q} \in A(\mathbf{p})$, then there exist $\mathbf{q}' \in A(\mathbf{p})$ such that $y(\mathbf{q}') = y'$ and $\mathbf{q}$ and $\mathbf{q}'$ are identical in all the parameter values except possibly y. An advancing algorithm A *fixes* a parameter y if A does not let y be free.

We will use T to indicate a Boolean algorithm or a set constraint that always returns true. As a short hand notation, if we omit a Boolean algorithm or a set constraint from a rule, then by that we mean that T is used.

We say that a pair of generalized positions $(\mathbf{p},\ \mathbf{q})$ is a **step** (with respect to a H-ASP program P) if there exists an advancing rule "$a \leftarrow B : A, O$" in P such that $\mathbf{p} \in O$ and $\mathbf{q} \in A(\mathbf{p})$. Then we will say that $\mathbf{p}$ is a **source** and $\mathbf{q}$ is a **destination**. We will assume that the underlying parameter space of P contains a parameter $Prev$. For a step $(\mathbf{p},\ \mathbf{q})$, then we will have $Prev(\mathbf{q}) = (x_1(\mathbf{p}), ..., \ x_n(\mathbf{p}))$. We define the advancing algorithm $GeneratePrev$ as $GeneratePrev(\mathbf{p}) = \{\mathbf{q}|$ $q \in S$ and $Prev(\mathbf{q}) = (x_1(\mathbf{p}), ..., \ x_n(\mathbf{p}))\}$ for a generalized position $\mathbf{p}$. (This can be implemented efficiently using references, rather than copies of data). We can then define a Boolean algorithm $isStep(\mathbf{p},\ \mathbf{q})$ equal true iff
$Prev(\mathbf{q}) = (x_1(\mathbf{p}), ..., \ x_n(\mathbf{p}))$ and $[t(\mathbf{q}) = t(\mathbf{p}) + stepSize]$.

For two one-place Boolean algorithms A, B, the notation $A \vee B$, $A \wedge B$, or $\overline{A}$ means a Boolean algorithm that for every generalized position $\mathbf{q}$ returns $A(\mathbf{q}) \vee B(\mathbf{q})$, $A(\mathbf{q}) \wedge B(\mathbf{q})$ or *not* $A(\mathbf{q})$ respectively. The same holds for two place Boolean algorithms.

For a domain state at time t we will assume that subsequent action states are at time $t + 0.1$ and the subsequent domain states are at time $t + 1$. Thus, advancing algorithms executing in domain states will increment time by 0.1, and advancing algorithms executing in action states will increment time by 0.9.

Finally, for a set of atoms M, $rules(M)$ will denote a set of stationary-1 rules $\{m \leftarrow : T \mid m \in M\}$.

2 Action Language Hybrid $\mathcal{AL}$

In this section, we shall define Hybrid $\mathcal{AL}$. Our definitions mirror the presentation of $\mathcal{AL}$ given in [10].

Syntax. In Hybrid $\mathcal{AL}$, there are two types of atoms: **domain atoms** and **action atoms**. There are two sets of parameters: **domain parameters** and **time**. The domain atoms are partitioned into three sorts: **inertial**, **static** and **defined**. A **domain literal** is a fluent atom p or its negation $\neg p$. For a generalized position $\mathbf{q}$, let $\mathbf{q}|_{domain}$ denote a vector of domain parameters.

A **domain algorithm** is a Boolean algorithm P such that for all the generalized positions $\mathbf{q}$ and $\mathbf{r}$, if $\mathbf{q}|_{domain} = \mathbf{r}|_{domain}$, then $P(\mathbf{q}) = P(\mathbf{r})$. An **action algorithm** is an advancing algorithm A such that for all $\mathbf{q}$ and for all $\mathbf{r} \in A(\mathbf{q})$ $time(\mathbf{r}) = time(\mathbf{q}) + 0.9$. For an action algorithm A, the signature of A, $sig(A)$, is the vector of parameter indices $i_0, i_1, ..., i_k$ of domain parameters such that A fixes parameters $i_0, i_1, ..., i_k$.

Hybrid $\mathcal{AL}$ allows the following types of statements.
1. **Action association statements**: *associate* a *with* A,
2. **Signature statements**: A *has signature* $i_0, ..., i_k$,
3. **Causal laws**: a *causes* $\langle l_{nd},\ L\rangle$ if $p_0, ..., p_m : P$,
4. **State constraints**: $\langle l,\ L\rangle$ if $p_0, ..., p_m : P$,
5. **Executability conditions**: *impossible* $a_0, ..., a_k$ if $p_0, ..., p_m : P$, and
6. **Compatibility conditions**: *compatible* $a_0,\ a_1$ if $p_0, ..., p_m : P$

where a is an action, A is an action algorithm, $i_0, ..., i_k$ are parameter indices, l_{nd} is a literal formed by an inertial or a static atom, L is a domain algorithm, $p_0, ..., p_m$ are domain literals, P is a domain algorithm, l is a domain literal, and $a_0, ..., a_k$ are actions $k \geq 0$ and $m \geq -1$. No negation of a defined fluent can occur in the heads of state constraints.

The following short-hand notation can be used for convenience.
(a) If L or P are omitted then the algorithm T is assumed to be used.
(b) For an action a, if the action association statement is omitted, then the action association statement "*associate* a *with* $\mathbf{0}$" is implicitly used for a. Here, $\mathbf{0}$ is an action algorithm with an empty signature that for a generalized position $\mathbf{p}$, produces the set of all generalized positions $\{\mathbf{q} : time(\mathbf{q}) = time(\mathbf{p}) + 0.9\}$.

Semantics. Similarly to $\mathcal{AL}$, a system description SD in Hybrid $\mathcal{AL}$ serves as a specification of the hybrid transition diagram $\mathcal{T}(SD)$ defining all possible trajectories of the corresponding dynamic system. Hence, to define the semantics of Hybrid $\mathcal{AL}$, we will define the states and the legal transitions of this diagram.

The H-ASP programs discussed below assume the parameter space consisting of parameters t (time), domain parameters and the parameter Prev. Such a parameter space will be called **the parameter space of** SD.

States. We let $\Pi_c(SD)$ be the logic program defined as follows.
1. For every state constraint $\langle l, L\rangle$ if $p_0, ..., p_m : P$, $\Pi_c(SD)$ contains the clause $l \leftarrow p_0, ..., p_m : P \vee \overline{L}$.
2. For every defined domain atom f, $\Pi_c(SD)$ contains the closed world assumption (CWA): $\neg f \leftarrow not\ f$.

For any set σ of domain literals, we let σ_{nd} denote the collection of all domain literals σ formed by inertial domain atoms and statics.

Definition 1. *Let* $(\sigma, \mathbf{q})$ *be a hybrid state. If* σ *is a complete and consistent set of domain literals, then* $(\sigma, \mathbf{q})$ *is a state of the hybrid transition diagram defined by a system description* SD *if* $(\sigma, \mathbf{q})$ *is the unique answer set of the program* $\Pi_c(SD) \cup rules(\sigma_{nd})$ *with the initial condition* $\mathbf{q}$.

The definitions of the fluent dependency graph, weak acyclicity, sufficient conditions for well-foundedness are the same as those defined for $\mathcal{AL}$.

Transitions. To describe a transition $\langle(\sigma_0, \mathbf{p}),\ a,\ (\sigma_1, \mathbf{q})\rangle$ we construct a program $\Pi(SD,\ (\sigma_0, \mathbf{p}),\ a)$, which as a similarly named program for $\mathcal{AL}$, consists of a logic program encoding of the system description SD, initial state $(\sigma_0, \mathbf{p})$ and an action a such that the answer sets of this program determine the states the system can move into after execution of a in $(\sigma_0, \mathbf{p})$.

The encoding $\Pi(SD)$ of the system description SD consists of the encoding of the signature of SD and rules obtained from the statements of SD. *The encoding of the signature* $sig(SD)$ into a set of stationary-1 rules is as follows:
(A) for each constant symbol c of sort *sort_name* other than fluent, static or action $sig(SD)$ contains: $sort\_name(c) \leftarrow$,
(B) for every defined fluent f of SD, $sig(SD)$ contains: $fluent(defined,\ f) \leftarrow$,
(C) for every inertial fluent f of SD, $sig(SD)$ contains: $fluent(inertial,\ f) \leftarrow$,
(D) for each static f of SD, $sig(SD)$ contains: $static(f) \leftarrow$, and
(E) for every action a of SD, $sig(SD)$ contains: $action(a) \leftarrow$

Next will specify *the encoding of statements of* SD.
(1) *For every action algorithm* A, we have an atom $\mathrm{alg}(A)$. If A has signature $(i_0, ..., i_k)$, then we add the following rules for all $j \in \{0, ..., k\}$ that specify all the parameters fixed by A: $fix\_value(i_j) \leftarrow action\_state,\ exec(\mathrm{alg}(A))$.
(2) *Inertia axioms for parameters*: For every domain parameter i, we have an advancing rule $discard \leftarrow action\_state,\ not\ fix\_value(i) :\ Default\,[i]$
where $Default\,[i]\,(\mathbf{p}) = \{\mathbf{q}|\ \mathbf{p}_i = \mathbf{q}_i\}$. The inertia axioms for parameters will cause the values of parameters not fixed by one of the action algorithms to be copied to the consequent states. *discard* here is a placeholder atom.
(3) *For every causal law*: a causes $\langle l_{in}, L\rangle$ if $p_0, ..., p_m : P$, $\Pi(SD)$ contains
(i). A stationary-1 rule generating atom $exec(\mathrm{alg}(A))$ specifying that algorithm A associated with action a will be used:
$exec(\mathrm{alg}(A)) \leftarrow action\_state,\ occurs(a),\ h(p_0), ..., h(p_m) : P.$
(ii) An advancing rule executing A to compute changes to domain parameters:
$h(l_{in}) \leftarrow action\_state,\ occurs(a),\ h(p_0), ..., h(p_m) : A, P.$
(iii) A stationary-2 rule to apply L to the successor states:
$h(l_{in}) \leftarrow action\_state, occurs(a), h(p_0), ..., h(p_m); : isStep \wedge [source(P) \vee dest(\overline{L})].$
(4) *For a one-place boolean algorithm* D, $source(D)$ indicates a two-place Boolean algorithm $source(D)(\mathbf{p}, \mathbf{q}) = D(\mathbf{p})$, and $destination(D)(\mathbf{p}, \mathbf{q}) = D(\mathbf{q})$.
(5) *For every state constraint*: $\langle l, L\rangle$ if $p_0, ..., p_m : P$, $\Pi(SD)$ contains
$h(l) \leftarrow domain\_state,\ h(p_0), ..., h(p_m) : P \vee \overline{L}.$
(6) $\Pi(SD)$ *contains CWA for defined fluents*
$\neg holds(f) \leftarrow domain\_state,\ fluent(defined, f),\ not\ holds(f).$
(7) *For every executability condition*: impossible $a_0, ..., a_k$ if $p_0, ..., p_m : P$, $\Pi(SD)$ contains
$\neg occurs(a_0) \vee ...\ \vee \neg occurs(a_k) \leftarrow action\_state,\ h(p_0),\ ...,\ h(p_m) : P.$
(8) $\Pi(SD)$ *contains inertia axioms for inertial fluents.* That is, for every inertial fluent f stationary-2 rules,
$holds(f) \leftarrow fluent(inertial, f),\ holds(f); not\ \neg holds(f) : isStep$ and
$\neg holds(f) \leftarrow fluent(inertial, f),\ \neg holds(f);\ not\ holds(f) :\ isStep.$
(9) $\Pi(SD)$ *contains propagation axioms for static and defined fluents.* These are used to copy statics and defined fluents from domain states to the successor action states. For every static or defined fluent f stationary-2 rules
$holds(f) \leftarrow domain\_state,\ static(f),\ holds(f);\ : isStep,$
$\neg holds(f) \leftarrow domain\_state,\ static(f),\ \neg holds(f);\ :\ isStep,$
$holds(f) \leftarrow domain\_state,\ fluent(defined, f),\ holds(f);\ : isStep$, and

$\neg holds(f) \leftarrow domain\_state,\ fluent(defined, f),\ \neg holds(f);\ :\ isStep.$
(10) *$\mathit{\Pi}(SD)$ contains CWA for actions*: for every every action a, there is a clause $\neg occurs(a) \leftarrow action\_state,\ not\ occurs(a)$.
(11) *For every action algorithm $A,\ B$* such that $sig(A) \cap sig(B) \neq \emptyset$, we have a stationary-2 rule prohibiting executing the algorithms in the same state, unless they are explicitly marked as compatible
$fail \leftarrow action\_state,\ exec(\text{alg}(A)),\ exec(\text{alg}(B)),$
$not\ compatible(\text{alg}(A),\ \text{alg}(B))$, and
$not\ compatible(\text{alg}(B),\ \text{alg}(A));\ not\ fail :\ isStep.$
(12) *For every compatibility condition*: compatible $a_0,\ a_1$ if $p_0, ..., p_m\ :\ P$, $\mathit{\Pi}(SD)$, there is a clause
$compatible(alg(A_0), alg(A_1)) \leftarrow action\_state, h(p_0), ..., h(p_m),$
$occurs(a_0), occurs(a_1) : P$
where A_i is the action algorithm associated with a_i.
(13) *$\mathit{\Pi}(SD)$ has axioms for describing the interleaving of domain and action states.* These are a stationary-2 rule and an advancing rule:
$domain\_state \leftarrow action\_state;\ :$
$action\_state \leftarrow domain\_state : CreateActionState$
where for a generalized position $\mathbf{p}$,
$CreateActionState(\mathbf{p}) =$
$\{\mathbf{q}|$ where $\mathbf{p}|_{domain} = \mathbf{q}|_{domain}\ \&\ time(\mathbf{q}) = time(\mathbf{p}) + 0.1\}$.
(14) *$\mathit{\Pi}(SD)$ contains rules making an action state with no actions invalid:* For every action a:
$valid\_action\_state \leftarrow action\_state,\ occurs(a)$
and a rule to invalidate the state without actions:
$fail \leftarrow action\_state,\ not\ valid\_action\_state,\ not\ fail.$
(15) *$\mathit{\Pi}(SD)$ contains the rule for generating the value of* Prev **parameter**:
$discard \leftarrow \text{GeneratePrev}, T.$

The inertia axioms and the propagation axioms guarantee that the set of fluents of a domain state and its successor action state are identical. The inertial axioms also guarantee that an action state and its successor domain state contain the same inertial fluents, if those fluents are not explicitly changed by causal laws.

A relation $holds(f, \mathbf{p})$ will indicate that a fluent f is true at a generalized position $\mathbf{p}$. $h(l, \mathbf{p})$ where l is a domain literal will denote $holds(f, \mathbf{p})$ if $l = f$ or $\neg holds(f, \mathbf{p})$ if $l = \neg f$. $occurs(a, \mathbf{p})$ will indicate that action a has occurred at $\mathbf{p}$. The encoding $h(\sigma_0, \mathbf{p})$ of the initial state is a set of stationary-1 rules:
$h(\sigma_0,\ \mathbf{p}) = \{h(l) \leftarrow: isDomainTime[t(\mathbf{p})] \mid l \in \sigma_0\}$
where $isDomainTime[x](\mathbf{q})$ returns true iff $t(\mathbf{q}) = x$.

Finally the encoding $occurs(a,\ \mathbf{p})$ of the action a is
$occurs(a,\ \mathbf{p}) = \{occurs(a_i) \leftarrow: isActionTime[t(\mathbf{p})] \mid a_i \in a\}$
where $isActionTime[x](\mathbf{q})$ returns true iff $t(\mathbf{q}) = x + 0.1$.

We then define $\mathit{\Pi}(SD,\ (\sigma_0, \mathbf{p}),\ a) = \mathit{\Pi}(SD) \cup h(\sigma_0,\ \mathbf{p}) \cup occurs(a,\ \mathbf{p})$.

Definition 2. *Let a be a nonempty collection of actions and $(\sigma_0, \mathbf{p})$ and $(\sigma_1, \mathbf{q})$ be two domain states of the hybrid transition diagram $\mathcal{T}(SD)$ defined by a system description $\mathcal{SD}$. A state-action-state triple $\langle(\sigma_0, \mathbf{p}),\ a,\ (\sigma_1, \mathbf{q})\rangle$ is a **transition***

of $\mathcal{T}(SD)$ iff $\mathit{\Pi}(\mathcal{SD},\ (\sigma_0, \mathbf{p}),\ a)$ has a stable model M with the initial condition $\mathbf{p}$*, such that $(\sigma_1, \mathbf{q})$ is a hybrid state.*

Hybrid $\mathcal{AL}$ provides a superset of the functionality of $\mathcal{AL}$. We prove this by defining a translation of a description SD in $\mathcal{AL}$ into $E(SD)$, which is a description in Hybrid $\mathcal{AL}$. We will then show that there is a correspondence between the states and transitions of $\mathcal{T}(SD)$ with those of $\mathcal{T}(E(SD))$.

The signature of $E(SD)$ contains exactly the domain and action atoms of SD, and no domain parameters. For every action law: "a causes l_{in} if p_0, ..., p_m" of SD, $E(SD)$ contains an action law "a causes (l_{in}, T) if $p_0, ..., p_m : T$". For every state constraint: "l if $p_0, ..., p_m$" of SD, $E(SD)$ contains a state constraint "(l, T) if $p_0, ..., p_m : T$". Finally, for every executability condition: "impossible $a_0, ..., a_k$ if $p_0, ..., p_m$" of SD, $E(SD)$ contains an executability condition: "impossible $a_0, ..., a_k$ if $p_0, ..., p_m : T$". We then have the following two equivalence theorems.

Theorem 1. *Let SD be a system description in action language $\mathcal{AL}$, and let σ be a complete and consistent set of domain literals. Then σ is a state of a transition diagram $T(SD)$ iff for all generalized positions* $\mathbf{q}$ *from the parameter space of $E(SD)$ $(\sigma,\ \mathbf{q})$ is a state of the hybrid transition diagram $T(E(SD))$.*

Theorem 2. *Let SD be a system description in action language $\mathcal{AL}$. If a state-action-state triple (σ_0, a, σ_1) of $T(SD)$ is a transition of $T(SD)$, then for all generalized positions $\mathbf{q}_0$ and $\mathbf{q}_1$ such that $\mathbf{q}_0|_{domain} = \mathbf{q}_1|_{domain}$ and $t(\mathbf{q}_0) + 1 = t(\mathbf{q}_1)$ from the parameter space of $E(SD)$, $((\sigma_0, \mathbf{q}_0),\ a,\ (\sigma_1, \mathbf{q}_1))$ is a transition of $T(E(SD))$. Moreover, if a state-action-state triple $((\sigma_0, \mathbf{q}_0),\ a,\ (\sigma_1, \mathbf{q}_1))$ of $T(E(SD))$ is a transition of $T(E(SD))$ then (σ_0, a, σ_1) is a transition of $T(SD)$.*

Sketch of a proof. First we construct the translation $\mathit{\Pi}(SD, \sigma_0, a)$ of SD and a similar translation $\mathit{\Pi}(E(SD), (\sigma_0, \mathbf{q}_0), a)$ of $E(SD)$. There is an equivalence of the one step provability operators of the two Gelfond-Lifschitz transforms with respect to domain atoms. Using this, one can define a bijection between the set of stable models specifying transitions $\{(\sigma_0, a, \sigma_1)\}$ of $T(SD)$, and the set of stable models specifying transitions $\{((\sigma_0, \mathbf{q}_0),\ a,\ (\sigma_1, \mathbf{q}_1))\}$ of $\mathit{\Pi}(E(SD), (\sigma_0, \mathbf{q}_0), a)$.

3 Example

We will now revisit the example from Figure 1 and describe it in Hybrid $\mathcal{AL}$. In our domain, the only action is *selectVideo.* The algorithm *selectVideoAlg* will produce a set of possible videos. Each such video will be stored in the parameter *video* of a possible state. The selected video quality will be checked by the domain algorithm *checkQuality.*

Our specification of actions has two statements:
associate selectVideo with selectVideoAlg and
selectVideoAlg has signature video.
In addition, there is a causal law,
selectVideo causes selected if not selected, and a state constraint,
malfunction if selected: -checkQualityAlg.

The translation of our specification into H-ASP is as follows.

```
% The encoding of the signature and action declaration:
fluent(inertial, selected):- . fluent(defined, malfunction):- . action(selectVideo):-
fix_value(video):- exec(alg(selectVideAlg))

% Inertia axioms for parameters
discard:- action_state, not fix_value(video): Default[video]

% Causal laws for: selectVideo causes selected if not selected
exec(alg(selectVideoAlg)):- action_state, occurs(selectVideo), -holds(selected)
holds(selected):- action_state, occurs(selectVideo), -holds(selected): selectVideoAlg, T
holds(selected):- action_state, occurs(selectVideo), -holds(selected);
      : isStep && [source(T) || dest(-T)]

% State constraint: malfunction if selected: -checkQualityAlg
holds(malfunction):- domain_state, holds(selected): -checkQualityAlg || -T

% CWA for the defined fluent:
-holds(malfunction):- domain_state, fluent(defined, malfunction), not holds(malfunction)

% Inertia axioms for the inertial fluent:
holds(selected):- fluent(inertial, selected), holds(selected); not -holds(selected);: isStep
-holds(selected):- fluent(inertial, selected), -holds(selected); not holds(selected);: isStep

% Propagation axioms: for defined fluents and CWA for the action:
holds(malfunction):- domain_state, fluent(defined, malfunction), holds(malfunction);: isStep
-holds(malfunction):- domain_state, fluent(defined, malfunction), -holds(malfunction);: isStep
-occurs(selectVideo):- action_state, not occurs(selectVideo)

% Interleaving of action and domain states, rules for generating action states:
domain_state:- action_state; : . action_state:- domain_state: CreateActionState

% Invalidate an action state with no actions; the rule for generating Prev parameter
valid_action_state:- action_state, occurs(selectVideo)
fail:- action_state, not valid_action_state, not fail
discard:- : GeneratePrev, T
```

We can now simulate our domain. A generalized positions will be written as a vector of 3 elements (*time*, *video*, *Prev*). The initial hybrid state has the generalized position $\mathbf{p}_0 = (0, \emptyset, ())$ (where $\emptyset$ is the initial value indicating that no data is available). The initial hybrid state can be encoded as:

```
domain_state:- : isDomainTime[0]
-holds(selected):- : isDomainTime[0] . -holds(malfunction):- : isDomainTime[0]
```

it is not difficult to see that it is a valid state according to our definitions.
The action $selectVideo$ executed at time 0 can be encoded as:

```
occurs(selectVideo):- : isActionTime[0]
```

We will assume that *selectVideoAlg* returns two videos: v1 and v2, and that checkQualityAlg succeeds on v1 and fails on v2. For brevity we will omit atoms specifying the signature, i.e. *fluent(inertial, selected)*, *fluent(defined, malfunction)* and *action(selectVideo)*, as these will be derived in every state. Our stable model will then consists of the following hybrid state encodings:

```
* Generalized position: (0, 0, ())
Atoms: -holds(selected). -holds(malfunction). domain_state

* Generalized position: (0.1, 0, Prev=(0))
Atoms: -holds(selected). -holds(malfunction). action_state. discard
   occurs(selectVideo). exec(alg(selectVideo)). valid_action_state. fix_value(video)

* Generalized position: (1, v1, Prev=(0))
Atoms: holds(selected). -holds(malfunction). domain_state. discard

* Generalized position: (1, v2, Prev=(0))
Atoms: holds(selected). holds(malfunction). domain_state. discard
```

4 Conclusions

In this paper we have introduced Hybrid $\mathcal{AL}$ - an extension of the action language $\mathcal{AL}$, that provides a mechanism for specifying both a transition diagram and associated computations for observing fluents and executing actions. This type of processing cannot be done easily with the existing action languages such as $\mathcal{AL}$ or $\mathcal{H}$ [6] without extending them. We think, however that this capability will be useful for improving computational efficiency in applications such as diagnosing malfunctions of large distributed software systems.

While the semantics of $\mathcal{AL}$ is defined using ASP, the semantics of Hybrid $\mathcal{AL}$ is defined using H-ASP - an extension of ASP that allows ASP type rules to control sequential processing of data by external algorithms. Hybrid $\mathcal{AL}$ and H-ASP can be viewed as part of the effort to expand the functionality of ASP to make it more useful along the lines of DLV^{DB} [13], VI [5], GRINGO [8] that allow interactions with the external data repositories and external algorithms. In [12], Redl notes that HEX programs [7] can be viewed as a generalization of these formalisms. Thus we will briefly compare H-ASP and HEX, as target formalisms for translating from a $\mathcal{AL}$-like action language. HEX programs are an extension of ASP programs that allow accessing external data sources and external algorithms via *external atoms*. The external atoms admit input and output variables, which after grounding, take predicate or constant values for the input variables, and constant values for the output variables. Through the external atoms and under the relaxed safety conditions, HEX programs can produce constants that don't appear in the original program.

There is a number of relevant differences between H-ASP and HEX. H-ASP has the ability to pass arbitrary binary information sequentially between external algorithm. While the same can be implemented in HEX, the restriction that the values of the output variables are constants means that a cache that uses constants as tokens, needs to be built around a HEX-based system to retrieve data. H-ASP also explicitly supports sequential processing required for $\mathcal{AL}$-like processing, whereas HEX does not. Such enhancements can also provide the potential for performance optimization. On the other hand, HEX allows constants returned by algorithms to be used as any other constants in the language. This gives HEX expressive advantage over H-ASP where the output of the algorithms cannot be easily used as regular constants.

In [4], the authors have described the use of H-ASP for diagnosing failures of Google's automatic whitelisting system for Dynamic Remarketing Ads, which is an example of a large distributed software system. The approach did not involve constructing a mathematical model of the diagnosed domain. The results of this paper can be viewed as a first step towards developing a solution to the problem of diagnosing malfunctions of a large distributed software system based on constructing a mathematical model of the diagnosed domain. The next steps in this development is to create a software system for Hybrid $\mathcal{AL}$.

Acknowledgments We would like to thank Michael Gelfond for insightful comments that helped to enhance the paper.

References

1. Balduccini, M., Gelfond, M.: Diagnostic reasoning with a-prolog. TPLP **3**(4-5) (2003) 425–461
2. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In: Logic Based Artificial Intelligence, Kluwer Academic Publishers. (2000) 257–279
3. Brik, A., Remmel, J.B.: Hybrid ASP. In Gallagher, J.P., Gelfond, M., eds.: ICLP (Technical Communications). Volume 11 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011) 40–50
4. Brik, A., Remmel, J.B.: Diagnosing automatic whitelisting for dynamic remarketing ads using hybrid ASP. In Calimeri, F., Ianni, G., Truszczynski, M., eds.: Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings. Volume 9345 of Lecture Notes in Computer Science., Springer (2015) 173–185
5. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Ann. Math. Artif. Intell. **50**(3-4) (2007) 333–361
6. Chintabathina, S., Gelfond, M., Watson, R.: Modeling hybrid domains using process description language. In Vos, M.D., Provetti, A., eds.: Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd Intl. ASP'05 Workshop, Bath, UK, September 27-29, 2005. Volume 142 of CEUR Workshop Proceedings., CEUR-WS.org (2005)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005, Professional Book Center (2005) 90–96
8. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: The potsdam answer set solving collection. AI Commun. **24**(2) (2011) 107–124
9. Gelfond, M., Inclezan, D.: Some properties of system descriptions of al_{d}. Journal of Applied Non-Classical Logics **23**(1-2) (2013) 105–120
10. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
11. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. (1988) 1070–1080
12. Redl, C.: Answer Set Programming with External Sources: Algorithms and Efficient Evaluation. PhD thesis, Vienna University of Technology (2015)
13. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **8**(2) (2008) 129–165
14. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at google with borg. In: Proceedings of the European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France, 2015. (2015)