Diagnosing Data Pipeline Failures Using Action Languages

Jori Bomanson[1] and Alex Brik[2]

[1] Department of Computer Science, Aalto University, Espoo, Finland
[2] Google Inc., Mountain View, USA

Abstract. This paper discusses diagnosis of industrial data processing pipelines using action languages. Solving the problem requires reasoning about actions, effects of the actions and mechanisms for accessing outside data sources. To satisfy these requirements, we introduce an action language, Hybrid $\mathcal{ALE}$ that combines elements of the action language Hybrid $\mathcal{AL}$ [6] and the action language C_{TAID} [8]. We discuss some of the practical aspects of implementing Hybrid $\mathcal{ALE}$ and describe an example of its use.

Answer Set Programming (ASP) is a knowledge representation formalism with the stable model semantics [11] that allows for a concise representation of defaults and uncertainty. Action languages [12] allow to formalize reasoning about effects of actions in dynamic domains. Descriptions in an action language are usually compiled into ASP. ASP solvers can then be used to find answer sets of the compiled descriptions, which specify possible trajectories of the modeled dynamic domain. Action languages have been used in various applications, such as planning [13], biological modeling [2], and diagnostic reasoning [1]. In this paper we discuss our work towards automating diagnosis of certain data processing pipelines at Google Inc. using action languages.

Industrial data processing pipelines can consists of hundreds of jobs, with outputs of some jobs consumed as inputs by others within the pipeline. In addition, pipelines themselves can have input dependencies on other pipelines. When working well, this architecture allows efficient and effective processing of large amounts of data. When a malfunction occurs, it can bring related data processing tasks to a halt, causing a set of cascading failures. The failures can cause an alert being dispatched to on-call engineers.

For the engineers, an alert presents a diagnostic challenge, as it can point to one of the later among the cascading failures, rather than an earlier one. The earlier causes have to be found before the underlying problem can be resolved thoroughly, and this can be tedious and time consuming. Moreover, multiple possible causes of failure may have to be investigated. Automating the diagnosing process can decrease the time required to fix failures. This can improve the fault tolerance of the system as well as decrease the workload for the engineers.

Earlier action languages, such as $\mathcal{AL}$ [3] focus on formalizing possible state-action-state transitions as well as applicability of actions. In diagnosing data processing pipelines, we found that reasoning also about the necessity of actions is conducive for creative adequate diagnostic software. $\mathcal{C}_{TAID}$ [8] is the earliest action language we found that provides all the needed language constructs.

In the context of our application, there remains, however a problem that $\mathcal{C}_{TAID}$ does not solve. In order to limit the number of possible diagnoses, it may be necessary to query outside sources for information about the diagnosed system (*outside information*). Such outside information may be, for instance the completion status of a diagnosed job, or properties of temporary files created while the diagnosed pipeline was running. In some cases precomputing all of the outside information is impractical, because of the large amount of data required to address the needs of all plausible trajectories. A more practical approach is to query the outside environment during the inference, since as the inference progresses the set of the plausible trajectories decreases in size.

Standard ASP does not provide the ability to interact with the outside environment during inference. Since action languages such as $\mathcal{AL}$ or $\mathcal{C}_{TAID}$ are translated into ASP, their ability to interact with the outside environment is likewise limited. There are extensions of ASP, however that provide the needed functionality. These include DLV^{DB} [14], VI programs [7], GRINGO grounder [10], HEX [9] and Hybrid ASP (H-ASP) [4]. The only action language known to us that translates into one of these extensions is Hybrid $\mathcal{AL}$ [6], which translates into H-ASP. We introduce a new action language Hybrid $\mathcal{ALE}$, which extends Hybrid $\mathcal{AL}$ with the language constructs of $\mathcal{C}_{TAID}$. We then discuss the use of Hybrid $\mathcal{ALE}$ for diagnostic reasoning in our application.

In this paper we refer to an example data processing pipeline and focus on a single job *process_data1* that requires an input file *input.data* before it starts processing and is suspended until the input file is produced. Upon successful termination, the job produces a single output file, *output_<timestamp>.data*, which can then be used as an input into a next job in the data processing pipeline. The *timestamp* is the timestamp of the output file and is part of its name. Upon a malfunction, the output file is not created.

To provide a meaningful diagnosis, in our example it is necessary to determine whether the input file and the output files exist. This can be done outside of the system description. In general, however such precomputing may not be feasible because of the large number of the possible trajectories of the diagnosed system. During inference, however, the space of the possible trajectories decreases as some possible trajectories are found invalid. Thus, if the checks are performed as needed during the inference, the computations can become more feasible.

The rest of the paper is structured as follows. In Section 1 we review H-ASP. In Section 2 we define Hybrid $\mathcal{ALE}$. In Section 3 we discuss the compilation of Hybrid $\mathcal{ALE}$ descriptions into H-ASP programs. A theorem demonstrating the correctness of the translation is discussed in the same section. In section 4 we discuss an example of the use of Hybrid $\mathcal{ALE}$ for diagnostic reasoning. A discussion of some of the practical aspects of implementing Hybrid $\mathcal{ALE}$ is in Section 5 followed by the conclusion.

1 Hybrid ASP

We now give a brief overview of H-ASP restricted to rules used in this work. 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 $\mathbf{p}_i$ denote x_i for $i = 1, \ldots, m$. For convenience we name certain parameters, and use their names instead of their indexes so that for instance if a parameter i is named n we may use $n\left(\mathbf{p}\right)$ to mean $\mathbf{p}_i$.

A *literal* is an atom a or its negation $\neg a$. For a literal b we define $\overline{b} = \neg a$ if $b = a$ for some atom a, and $\overline{b} = a$ if $b = \neg a$ for some atom a.

The *universe* of P is $At \times S$. A pair $(Z, \mathbf{p})$ where $Z \subseteq At$ and $\mathbf{p} \in S$ is 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 $B = a_1, \ldots, a_n$, *not* $b_1, \ \ldots,$ *not* b_m where $a_1, \ldots, a_n, b_1, \ldots, b_m \in At$. We let $B^- =$ *not* $b_1, \ \ldots,$ *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 $\mathbf{q} \in A(\mathbf{p})$, $t(\mathbf{q}) > t(\mathbf{p})$. 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. A is called an *advancing algorithm.*

Stationary-*i* rules (for $i = 1$ or $i = 2$) are of the form: $a \leftarrow B_i; \ B_1 : H, O$ (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 $(\mathbf{p}_i, \mathbf{p}_1) \in O$ (where for $i = 1$ we mean $\mathbf{p}_1 \in O$), B_k is satisfied at $\mathbf{p}_k$ for $k = 1, i$, and $H(\mathbf{p}_i, \mathbf{p}_1)$ is true (where for $i = 1$ we mean $H(\mathbf{p}_1)$), then $(a, \ \mathbf{p}_i)$ holds. H is called a *predicate algorithm.*

The stable model semantics for H-ASP [4] defines a set of *answer sets* for an H-ASP program in terms of a reduct in a way similar to ASP, but it is omitted here due to space constraints.

We now introduce additional definitions which are used later in this paper. 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 that $\mathbf{q}$ and $\mathbf{q}'$ are identical in all parameter values except possibly y. An advancing algorithm A *fixes* a parameter y if A does not let y be free. Intuitively, A fixes y if A is intended to specify values for y, and A lets y be free otherwise.

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

A pair of generalized positions $(\mathbf{q}, \mathbf{p})$ 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 say that $\mathbf{p}$ is a *source* and $\mathbf{q}$ is a *destination.* We assume that the underlying parameter space of P contains a parameter *Prev* defined so that, for a step $(\mathbf{q}, \mathbf{p})$, we have $Prev(\mathbf{q}) = (x_1(\mathbf{p}), \ ..., \ x_n(\mathbf{p}))$. We also define a

Boolean algorithm $IsStep(\mathbf{p},\ \mathbf{q})$ that is true iff
$Prev(\mathbf{q}) = (x_1(\mathbf{p}),\ ...,\ x_n(\mathbf{p}))$ and $[t(\mathbf{q}) = t(\mathbf{p}) + stepSize]$.

Given one-place Boolean algorithms A, B, we write $A \vee B$, $A \wedge B$, and $\overline{A}$ for Boolean algorithms that map generalized positions $\mathbf{q}$ to $A(\mathbf{q}) \vee B(\mathbf{q})$, $A(\mathbf{q}) \wedge B(\mathbf{q})$, and *not* $A(\mathbf{q})$ respectively. The same holds for two-place Boolean algorithms.

For a set of literals M, we denote by $rules(M)$ and $defaults\,(M)$ the sets of stationary-1 rules $\{m \leftarrow: T \mid m \in M\}$ and $\{m \leftarrow not\ \overline{m} \mid m \in M\}$ respectively.

A notation of the form $\leftarrow a_1, ..., a_m\ :\ P$ stands for a *constraint*, i.e., a stationary-1 rule $\_fail \leftarrow a_1, ..., a_m,\ not\ \_fail : P$, where $\_fail$ is a new auxiliary atom. An analogous notation holds for a stationary-2 rule.

If we omit an advancing algorithm from an advancing rule, by that we mean an advancing rule where an algorithm A is used such that if $\mathbf{q} \in A\,(\mathbf{p})$ then $(\mathbf{q}, \mathbf{p})$ is a step, and A lets all the parameters except *time* and *Prev* be free.

An algorithm $Dest\,[P]$ is defined to hold for $\mathbf{p}, \mathbf{q}$ iff $P\,(\mathbf{p})$.

2 Action Language Hybrid $\mathcal{ALE}$

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 edge labels are subsets of actions. An edge indicates that simultaneous execution of the actions in the label of an edge can transform a source state into a destination state. The transformation is not necessarily deterministic, and for a given source state there can be multiple edges having different destination states, labeled with the same set of actions. In Hybrid $\mathcal{ALE}$, just as 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 Boolean properties of a system, and $\mathbf{p}$ as a 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 $\mathbf{p} = \mathbf{r}$. We note that there is a simple bijection between the set of transition diagrams and the set of hybrid transition diagrams.

We now define Hybrid $\mathcal{ALE}$ **syntax.** In Hybrid $\mathcal{ALE}$, there are two types of atoms: *fluents* and *actions.* There are two types of parameters: *domain parameters* and *time.* The fluents are partitioned into *inertial* and *default.* A *domain literal* l is a fluent atom p or its negation $\neg p$. For a generalized position $\mathbf{q}$, we let $\mathbf{q}|_{domain}$ denote a vector of domain parameters. The domain parameters are partitioned into *inertial* and *default.*

A *domain algorithm* is a Boolean algorithm P such that for all 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 $\mathbf{r} \in A(\mathbf{q})$, $time(\mathbf{r}) =$

$time(\mathbf{q})+1$. For an action algorithm A, the signature of A, $sig(A)$, is the vector of parameter indices $i_1, ..., i_k$ of domain parameters fixed by A.

Hybrid $\mathcal{ALE}$ allows the following types of statements.

1. **Default declaration for fluents:** *default fluent* l
2. **Default declaration for parameters:** *default parameter* i *with value* w
3. **Causal laws**: a *causes* $\langle l,\ L\rangle$ with A if $p_0, ..., p_m : P$,
4. **State constraints**: $\langle l,\ L\rangle$ if $p_0, ..., p_m : P$,
5. **Noconcurrency condition**: *impossible* $a_0, ..., a_k$ if $p_0, ..., p_m : P$,
6. **Allow condition:** *allow* a if $p_0, ..., p_m : P$,
7. **Trigger condition:** *trigger* a if $p_0, ..., p_m : P$,
8. **Inhibition condition:** *inhibit* a if $p_0, ..., p_m : P$

where l is a domain literal, i is a parameter index, w is a parameter value, a is an action, A is an action algorithm, $i_0, ..., i_k$ are parameter indices, L and P are domain algorithms, $p_0, ..., p_m$ are domain literals, and $a_0, ..., a_k$ are actions $k \geq 0$ and $m \geq -1$. If L or P are omitted then the algorithm T is substituted.

A *default declaration for fluents* declares a default fluent and specifies its default value. If l is a positive literal, then the default value is *true*, and if l is a negative fluent then the default value is *false*. A *default declaration for parameters* declares that i is a default parameter and that w is its default value. A *causal law* specifies that if p_0, ..., p_m hold and P is true when a occurs, then l holds and L is true after the occurrence of a. In addition, after a occurs, the values of the parameters $sig(A)$ are specified by the output of the action algorithm A. A *state constraint* specifies that whenever p_0, ..., p_m hold and P is true, l also holds and L is true. A *noconcurrency condition* specifies that whenever $p_0, ..., p_m$ hold and P is true, $a_0, ..., a_k$ cannot occur concurrently pairwise. An *allow condition* specifies that whenever p_0, ..., p_m hold and P is true, an action a can occur (although not necessarily so). A *trigger condition* specifies that whenever p_0, ..., p_m hold and P is true, an action a necessarily occurs (unless inhibited). An *inhibition condition* specifies that whenever p_0, ..., p_m hold and P is true, action a cannot occur. A *system description SD* is a set of Hybrid $\mathcal{ALE}$ statements.

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 is called *the parameter space of SD*.

Let $\Pi_c(SD)$ denote the logic program: for every state constraint of the form (4), $\Pi_c(SD)$ contains the rules $l \leftarrow p_0, ...,\ p_m : P$ and $\leftarrow p_0, ...,\ p_m : P \wedge \overline{L}$.

Definition 1. *Let $(\sigma, \mathbf{q})$ be a hybrid state, and let $\sigma' \subseteq \sigma$ and $\sigma'' \subseteq \sigma$. If σ is a complete and consistent set of domain literals, then $(\sigma, \mathbf{q})$ is a Hybrid $\mathcal{ALE}$ state relative to σ', σ'' if $(\sigma, \mathbf{q})$ is an answer set of the program $\Pi_c(SD) \cup rules(\sigma') \cup defaults\,(\sigma'')$ with the initial condition $\mathbf{q}$.*

Next, we introduce a number of definitions needed to specify a transition.

A causal law or a state constraint is *applicable* in $(\sigma, \mathbf{q})$ if $\{p_0, ...,\ p_m\} \subseteq \sigma$ and $P\,(\mathbf{q})$ holds. The *logical effects* of an action a in a state $(\sigma, \mathbf{q})$ are $LE((\sigma, \mathbf{q}), a) = \{l : (a\ causes\ \langle l, L\rangle\ with\ A\ if\ p_0, ...,\ p_m : P)$ is applicable in $(\sigma, \mathbf{q})\}$. For a set of actions B we define $LE\,((\sigma, \mathbf{q})\,, B) = \bigcup_{a \in B} LE\,((\sigma, \mathbf{q})\,, a)$.

For a generalized position $\mathbf{q}$ and action algorithm A with signature $(i_1, ..., i_k)$ we define the *binary effects* of A in $\mathbf{q}$ as $BE(\mathbf{q}, A) = \{((i_1, r_1), ..., (i_k, r_k)) : (r_1, ..., r_k) \in A(\mathbf{q})\}$.

A tuple $u = ((i_1, r_1), ..., (i_k, r_k))$ where i_js are parameter indexes and r_js are the values of the corresponding parameters is called an *assignment tuple*. We define $sig(u) = (i_1, ..., i_k)$ and $values(u) = (r_1, ..., r_k)$.

The *binary effects* of a set of actions D, in a state $(\sigma, \mathbf{q})$ are $BE((\sigma, \mathbf{q}), D) = \{BE(\mathbf{q}, A) : (a \text{ causes } \langle l, L\rangle \text{ with } A \text{ if } p_0, ..., p_m : P)$ is applicable in $(\sigma, \mathbf{q})$ and $a \in D\}$.

For the binary effects of the actions B let $\Delta_p(B) = ((j_1, r_1), ..., (j_n, r_n))$ where $(j_1, ..., j_n)$ are the parameters not present among those that are in B, and for a parameter j_m, $r_m = \mathbf{q}_{j_m}$ if j_m is an inertial parameter, and $r_m = w$ if j_m is a default parameter with the default w. A *binary effects completion* of B is $\overline{B} = B \cup \{\ \Delta_p(B)\ \}$. That is a binary effects completion of B contains B and the assignment tuple for the parameters not found in B.

If w is an assignment tuple $((j_1, q_1), ..., (j_k, q_n))$ and $sig(u) \cap sig(w) = \emptyset$ then the *product* of u and w is an assignment tuple $((l_1, p_1), ..., (l_{k+n}, p_{k+n}))$ where $l_1, ..., l_{k+n}$ is the arrangement of the indexes $i_1, ..., i_k, j_1, ..., j_n$ in increasing order and $p_1, ..., p_{k+n}$ is the corresponding arrangement of the values.

For a set S of assignment tuples let $sig(S) = \{\ sig(u) : \ u \in S\}$. We say that S is *valid* if whenever $s_1, s_2 \in sig(S)$ are such that $s_1 \cap s_2 = \emptyset$ where the intersection of the two tuples is a tuple of the elements in the intersection of s_1, s_2 with s_1 and s_2 treated as sets.

For a valid set S of assignment tuples and a signature $s \in sig(S)$ we define $AT(s, S) = \{x : x \in S \text{ and } sig(x) = s\ \}$. A *partition of S by signatures* is $Part(S) = \{AT(s, S) : s \in sig(S)\ \}$.

The *set of the candidate successor generalized positions* at $(\sigma, \mathbf{q})$ with respect to a set of actions D is

$CSGP((\sigma, \mathbf{q}), D) = \emptyset$ if $\overline{BE}((\sigma, \mathbf{q}), D)$ is not valid, and otherwise

$CSGP((\sigma, \mathbf{q}), D) =$
$values(\prod Part(\overline{BE}((\sigma, \mathbf{q}), D)\ \cup \{\ (0, t(\mathbf{q}) + 2)\ \}\ \cup\ \{\ (\text{Prev},\ \mathbf{q}|_{domain})\ \})).$

This specifies that given the binary effects of the action algorithms of the applicable causal laws, the candidate successor generalized positions can be constructed by taking the "cross products" of the binary effects of the corresponding action algorithms and by substituting any missing parameters i with $\mathbf{q}_i$ if i is an inertial parameter, or the default value of i if it is a default parameter.

For a state $(\sigma_0, \mathbf{q})$ and a set of actions B we define *the set of consequent states* as:

$CS((\sigma_0, \mathbf{q}),\ B) = \{\ (\sigma, \mathbf{r})\ : \mathbf{r} \in CSGP((\sigma_0, \mathbf{q}),\ B)$ and $L(\mathbf{r})$ holds for all L s.t. $(a \text{ causes } \langle l, L\rangle \text{ with } A \text{ if } p_0, ..., p_m : P)$ is applicable at $(\sigma_0, \mathbf{q})$ and $a \in B$ and $LE((\sigma_0, \mathbf{q}),\ B) \subseteq \sigma$ and $(\sigma, \mathbf{r})$ is a Hybrid $\mathcal{ALE}$ state relative to $LE((\sigma_0, \mathbf{q}),\ B)$, $\{l :\ l \in \sigma_0$ and l is inertial$\} \cup \{l : l$ is a default fluent$\}\ \}$.

That is the set of consequent states are constructed by combining the set of the candidate successor generalized positions with the Hybrid $\mathcal{ALE}$ states relative to the set of the logical effects of the applicable actions.

Finally, we specify the sets of possible and necessary actions similarly to [8]. An inhibition condition, an allow condition, a trigger condition and a noconcurrency condition is *active in* $(\sigma, \mathbf{q})$ if $\{p_0, ..., p_m\} \subseteq \sigma$ and $P(\mathbf{q})$ holds. Let $A_I(\sigma, \mathbf{q}) = \{a :$ there exists an active inhibition condition in SD containing $a\}$. Let $A_T(\sigma, \mathbf{q}) = \{a :$ there exists an active trigger condition in SD containing a and $a \notin A_I(\sigma, \mathbf{q})\}$. Let $A_A(\sigma, \mathbf{q}) = \{a :$ there exists an active allow condition in SD containing a and $a \notin A_I(\sigma, \mathbf{q})\}$. Let $A_N(\sigma, \mathbf{q}) = \{(a_1, ..., a_n) :$ there exists an active noconcurrency condition with $a_1, ..., a_n\}$.

Definition 2. *(Transition) Hybrid $\mathcal{ALE}$ states $(\sigma_0, \mathbf{q})$, $(\sigma_1, \mathbf{r})$ and a nonempty set of actions B form a transition of SD if $(\sigma_1, \mathbf{r}) \in CS((\sigma_0, \mathbf{q}), B)$ and $A_T(\sigma_0, \mathbf{q}) \subseteq B \subseteq A_T(\sigma_0, \mathbf{q}) \cup A_A(\sigma_0, \mathbf{q})$ and for all $B' \in A_N(\sigma_0, \mathbf{q})$ we have $|B \cap B'| \leq 1$.*

In a transition there is always a reason for an action occurring. The definition ensures that no inhibited action is included in B, that all the triggered actions that are not inhibited are in B, that the remaining actions in B are allowed and that actions prohibited from executing concurrently by a noconcurrency condition are not all in B.

3 Compilation

A system description SD in Hybrid $\mathcal{ALE}$ is compiled into H-ASP. In the definition below we assume that the Hybrid $\mathcal{ALE}$ statements are of the form (1)-(8) specified in the syntax definition. *The encoding $\Pi(SD)$* of the system description SD consists of the following:

1. For every action algorithm A, we have an atom $\text{alg}(A)$. If A has a signature $(i_0, ..., i_k)$ then we add the following rules for $j \in \{0, ..., k\}$ that specify all the parameters fixed by A and execute the algorithm A when appropriate:
a stationary-1 rule, $will\_fix(i_j) \leftarrow action\_state,\ exec(\text{alg}(A))$,
a stationary-2 rule, $fix(i_j) \leftarrow;\ action\_state,\ exec(\text{alg}(A))$,
an advancing rule, $domain\_state \leftarrow action\_state,\ exec(\text{alg}(A)) : A$.
For every pair of algorithms A_1 and A_2 with a nonempty signature intersection, we add the following stationary-1 rule,
$\leftarrow action\_state,\ exec(\text{alg}(A_1)),\ exec(\text{alg}(A_2))$, to prevent situations where two different algorithms are setting the values of the same parameter in the same state.
2. *Inertia axioms for parameters.* For every inertial domain parameter i, we have an advancing rule
$fix(i) \leftarrow action\_state,\ not\ will\_fix(i) :\ Default[i]$
where $Default[i](\mathbf{p}) = \{\mathbf{q} :\ \mathbf{p}_i = \mathbf{q}_i\ \}$. The inertia axioms for parameters cause the values of the inertial parameters not fixed by one of the action algorithms to be copied to the successor states.
3. *Default axioms for parameters.* For every default parameter i with the value w, we have an advancing rule
$fix(i) \leftarrow action\_state,\ not\ will\_fix(i) : Default[i, w]$

where $Default[i,w]\left(\mathbf{p}\right)=\{\mathbf{q}:\mathbf{p}_i=w\}$. The default axioms for parameters cause the values of the default parameters not fixed by one of the action algorithms to be set to the default value.

4. *State restriction constraints for the parameters.* We restrict the possible domain states to only those where every parameter is marked as fixed. For every domain parameter i, we have a stationary-1 rule
$\leftarrow not\ fix\left(i\right),\ domain\_state$
5. For every causal law $c \in SD$ of the form (3):
   (a) a stationary-2 rule specifying that the law is applicable if the prerequisites are satisfied
   $causal\left(c\right) \leftarrow\ action\_state,\ occurs\left(a\right),\ h\left(p_0\right),...,h\left(p_m\right):P$
   where $causal\left(c\right)$ is an atom uniquely identifying the causal law.
   (b) a stationary-1 rule specifying that the advancing algorithm A is to be evaluated if the prerequisites are satisfied,
   $exec\left(\mathrm{alg}\left(A\right)\right) \leftarrow causal\left(c\right)$
   (c) a stationary-2 rule to derive $h\left(l\right)$ in the successor state,
   $h\left(l\right) \leftarrow\ ;\ causal\left(c\right):IsStep$
   (d) a stationary-2 rule specifying that L must be true in the successor state,
   $\leftarrow\ ;\ causal\left(c\right):IsStep \wedge Dest[\overline{L}]$.
6. For every state constraint $s \in SD$ of the form (4):
   (a) a stationary-1 rule indicating that the state constraint is applicable
   $constraint\left(s\right) \leftarrow domain\_state,\ h\left(p_0\right),...,\ h\left(p_m\right):P$
   where $constraint\left(s\right)$ is an atom identifying s.
   (b) a stationary-1 rule to derive $h\left(l\right)$,
   $h\left(l\right) \leftarrow constraint\left(s\right),\ domain\_state$
   (c) and a stationary-1 rule to verify that L holds if the rule is applicable
   $\leftarrow constraint\left(s\right),\ domain\_state:\overline{L}$
7. For every noconcurrency condition $n \in SD$ of the form (5):
   (a) a stationary-1 rule indicating that the condition is applicable
   $noconcurrency\left(n\right) \leftarrow action\_state,\ h\left(p_0\right),...,\ h\left(p_m\right):P$
   (b) a stationary-1 rule for every pair $a_i,\ a_j \in \{a_0,...,a_k\}$ to make the concurrent occurrence of a_i and a_j impossible
   $\leftarrow occurs\left(a_i\right),\ occurs\left(a_j\right),\ noconcurrency\left(n\right)$
8. For every trigger condition of the form (7), a stationary-1 rule to trigger the occurrence of a
$occurs\left(a\right) \leftarrow action\_state,\ not\ ab\left(occurs\left(a\right)\right),\ h\left(p_0\right),...,\ h\left(p_m\right):P$
9. For every inhibition condition of the form (8), a stationary-1 rule to inhibit the occurrence of a
$ab\left(occurs\left(a\right)\right) \leftarrow action\_state,\ h\left(p_0\right),...,\ h\left(p_m\right):P$
10. For every allow condition of the form (6) to make the occurrence of the action a possible:
$allow\left(a\right) \leftarrow action\_state,\ h\left(p_0\right),...,\ h\left(p_m\right),\ not\ ab\left(occurs\left(a\right)\right):P$
$occurs\left(a\right) \leftarrow\ allow\left(a\right),\ not\ \neg occurs\left(a\right)$
$\neg occurs\left(a\right) \leftarrow allow\left(a\right),\ not\ occurs\left(a\right)$

11. Axioms for interleaving domain states and action states. A stationary-2 rule
$domain\_state \leftarrow ;\ action\_state : IsStep$
and an advancing rule
$action\_state \leftarrow\ domain\_state : CreateActionState$
where for a generalized position $\mathbf{p}$
$CreateActionState\,(\mathbf{p}) = \{\mathbf{q}$: where $\mathbf{p}|_{domain} = \mathbf{q}|_{domain}$ and $time\,(\mathbf{q}) = time\,(\mathbf{p}) + 1,\ \text{Prev}\,(\mathbf{q}) = \mathbf{p}|_{domain}\ \}$.
12. Stationary-1 rules for making an action state with no actions invalid:
a constraint, $\leftarrow action\_state,\ not\ valid\_action\_state$,
and for every action a the rule, $valid\_action\_state \leftarrow\ action\_state,\ occurs\,(a)$.
13. Rules for copying fluents from a domain state to an action state. For every fluent l, stationary-2 rules
$h\,(l) \leftarrow ;\ domain\_state,\ h\,(l) : IsStep$
14. For every inertial literal l, a stationary-2 rule encoding *inertia axioms*
$h\,(l) \leftarrow\ not\ \overline{h\,(l)};\ action\_state,\ h\,(l) : IsStep$
15. For every default literal l, a stationary-1 rule encoding the default
$h\,(l) \leftarrow domain\_state,\ not\ \overline{h\,(l)}$
16. $\Pi\,(SD)$ contains closed world assumptions (CWA, for short) for actions. For every action a, a stationary-1 rule
$\neg occurs\,(a) \leftarrow\ action\_state,\ not\ occurs\,(a)$

The encoding $H\,(\sigma_0)$ of the initial state is a set of stationary-1 rules:
$H\,(\sigma_0) = \{\ h\,(l) \leftarrow : IsTime\,[0]\ : l \in \sigma_0\} \cup \{\ domain\_state\ \leftarrow : IsTime\,[0]\}$, where $IsTime\,[0]\,(\mathbf{p})$ holds iff $t\,(\mathbf{p}) = 0$.

Theorem 1. *(correctness of the translation) Hybrid $\mathcal{ALE}$ states $(\sigma_0, \mathbf{q})$, $(\sigma_1, \mathbf{r})$ where $t\,(\mathbf{q}) = 0$ and a set of actions B is a transition of SD iff there exists a stable model M of $\Pi\,(SD) \cup H\,(\sigma_0)$ with respect to $\mathbf{q}$ such that $\{q, \mathbf{r}\} \subseteq GP\,(M)$*
and $\{\ l : h\,(l) \in W_M\,(\mathbf{q})\ \} = \sigma_0$
and $\{\ l : h\,(l) \in W_M\,(\mathbf{r})\ \} = \sigma_1$
and there exists $\mathbf{s} \in GP\,(M)$ with $IsStep\,(\mathbf{s}, \mathbf{q})$ and $IsStep\,(\mathbf{r}, \mathbf{s})$ holding (i.e. $\mathbf{s}$ is a successor generalized position of $\mathbf{q}$ and $\mathbf{r}$ is a successor generalized position of $\mathbf{s}$) and $\{\ a :\ occurs\,(a) \in W_M\,(\mathbf{s})\} = B$.

The proof of the forward direction is by constructing M and then applying induction on the reduct of $\Pi\,(SD) \cup H\,(\sigma_0)$ with respect to M and $\mathbf{q}$ to show that M is a stable model of $\Pi\,(SD) \cup H\,(\sigma_0)$ with respect to $\mathbf{q}$. The proof of the reverse direction uses induction on the reducts to show that a stable model of $\Pi\,(SD) \cup H\,(\sigma_0)$ is a transition. The proof is omitted due to space constraints.

4 Example

In this section, we will discuss the example described in the beginning of the paper. Note that in the example the output file has a timestamp dependent name, which therefore cannot be hard coded. We use an advancing algorithm to find the name and to store it under a parameter. The parameter value can then be used in subsequent diagnosis stages to check the existence of the file.

We have two actions *do(1)* and *fail(1)*. The action *do(1)* indicates the attempted execution of a job. The action *fail(1)* indicates a malfunction. We use a default parameter *input_filename1* to contain the name of the input file, and we use a default parameter *output_filename1* to contain the name of the output file. We use the default fluent *ready(1)* with the default value *false*, to indicate that the input file exists and that the processing can start. All other fluents are inertial. *finished(1)* indicates a completed processing, whether successful or not. *failed(1)* indicates that the processing has failed, and *succeeded(1)* indicates that the processing has succeeded. Because *ready(1)* is a default fluent with the default value false, we know that the fluent holds only at a specific time in the trajectory. Thus, we do not need to check the negative conditions when checking the existence of the fluent. The predicate algorithm *Exists[file_name]* returns true if the value of the parameter *file_name* is not empty, which indicates the existence of the corresponding file. We use an advancing algorithm *GetOutputFileName1* to determine the output file name if it exists, and to associate the file name (or empty value, if the file does not exist) with the parameter *output_filename1*.

We then use a state constraint to determine whether the processing can start:
(1) *ready(1)* **if** *-finished(1)* : *Exists[input_filename1]*

ready(1) triggers the processing with an optional failure:
(2) **trigger** *do(1)* ***if*** *ready(1)*
(3) **allow** *fail(1)* ***if*** *ready(1)*

Indicate the completion of the processing and determine the output file name, if the file exists.
(4) *do(1)* ***causes*** *finished(1)* **with** *GetOutputFileName1*
(5) *fail(1)* ***causes*** *failed(1)*

Define success as an absence of failure.
(6) *succeeded(1)* ***if*** *finished(1), -failed(1)*

Make it invalid to fail if an output file exists, and to succeed if it does not.
(7) *<failed(1), -Exists[output_filename1]>* ***if*** *failed(1)*
(8) *<succeeded(1), Exists[output_filename1]>* ***if*** *succeeded(1)*

We now consider the two trajectories described by the above system description. We assume that the input file exists, the parameter *input_filename1* at the generalized position **q** contains the input file name, and *-finished(1)* is derived at a generalized position **q**. We consider the case where the output file exists.

(1) derives (*ready(1),* **q**). Consequently (2) and the absence of active inhibition conditions for the action *do(1)* cause an action *do(1)* to occur at state corresponding to **q**. (3) is active and creates two transitions: one containing

action *fail(1)* and one not containing the action. In both transitions, (4) derives (*finished(1),* **r**). Since the output file exists, (4) makes the parameter *output_filename1* at the generalized position **r** contain the name of the file. (5) is not active in the first transition, but in the second transition it derives (*failed(1),* **r**). (6) derives (*succeeded(1),* **r**) in the first transition, and (6) is not active in the second transition. (7) is not active in the first transition, but it invalidates the second transition, since *-Exists[output_filename1]* returns false. (8) is active in the first transition, but it simply rederives (*succeeded(1),* **r**).

Then the following transition is derived (we omit negative atoms from the description for brevity):

({ *ready(1)* }, **q**), { *do(1)* }, ({*finished(1), succeeded(1)*}, **r**)
where *input_filename1(***q***)*, *output_filename1(**r**)* contain the name of the input file and the name of the output file respectively.

Since only one transition is valid, and it contains (*succeeded(1),* **r**), the diagnosing engineer can conclude that *process_data1* has not failed.

If the output file did not exist, then the following transition would be derived:
({ *ready(1)* }, **q**), { *do(1), fail(1)* }, ({*finished(1), failed(1)*}, **v**)
where *input_filename1(***q***)* contains the name of the input file,
and *output_filename1(**v**)* is empty.

Since only one transition is valid, and it contains (*failed(1), **r***), the diagnosing engineer can conclude that *process_data1* has failed.

In our example we consider a data processing job typical of those found in the data pipelines we have worked with. Such data pipelines may contain many jobs, with outputs of some being the inputs of others. Each such jobs requires its own Hybrid $\mathcal{AL}E$ description. To create a diagnostic description for the entire pipeline, individual job descriptions need to be assembled into a single pipeline description. In many cases doing so is simplified because the interactions of the jobs is limited to the following scenario: an output of a job X is an input to a job Y. We can thus "chain" the descriptions together by using *finished(X)* as a condition for *ready(Y)*. If Y starts only upon a successful termination of X then -*failed(X)* has to be added to the condition for *ready(Y)* as well.

5 Computation

While a detailed discussion of all the relevant computational aspects is outside of the scope of this paper, we would like to note a few here. To compute with the Hybrid ASP programs compiled from Hybrid $\mathcal{AL}E$ descriptions, supporting only advancing rules of arity 1, and stationary rules of arity 1 and arity 2 is required. Moreover, computations can be made more efficient by assuming that any stationary-2 rule are applicable only in the generalized positions that form a step. In [5] it was shown that with the above and some additional adaptations all of the maximal trajectories can be computed using the *Local Algorithm.* Informally, the algorithm does the following. For a given hybrid state $(A, \mathbf{p})$ it first computes a set $S = \{ \ (B, \ \mathbf{q}) \ \}$ of candidate successor states via the use of the advancing rules applicable at $(A, \mathbf{p})$. Then for each $(B, \mathbf{q}) \in S$ it uses stationary-2 rules applicable at $(B, \mathbf{q})$, $(A, \mathbf{p})$ and stationary-1 rules applicable at $(B, \mathbf{q})$ to compute a set of the successor states at the generalized position $\mathbf{q}$. This is an iterative process that produces a tree, with hybrid states $(A, \mathbf{p})$ as nodes having their successor states as children. The trajectories can then be recovered by following tree paths from leaf nodes to the root.

Another adaptation is a deferred evaluation of domain algorithms. Evaluation of the domain algorithms can be computationally expensive, and for stationary rules, it is thus desirable to evaluate domain algorithms only once the satisfaction of boolean atoms is verified. We illustrate the deferred domain algorithm evaluation using the example of a stationary-1 rule: $constraint\,(s) \leftarrow domain\_state,\ h\,(p_0)\,, ...,\ h\,(p_m) : P$. For stationary-2 rules the implementation is similar. We introduce two atoms $pos(domain(P))$ and $neg(domain(P))$ that encode the possible values of evaluating P. We also add an atom $exec(domain(P))$ to indicate domain algorithms P that need to be evaluated. We then substitute the above stationary-1 rule with the following rules.

$constraint\,(s) \leftarrow domain\_state,\ h\,(p_0)\,, ...,\ h\,(p_m)\,,\ pos(domain(P))$
to indicate that $constraint\,(s)$ can be derived only if all the predicate requirements are satisfied and P evaluates to true. We add a rule to indicate that P needs to be evaluated if all the propositional constraints of the original stationary-1 rule are satisfied:

$exec(domain(P)) \leftarrow domain\_state,\ h\,(p_0)\,, ...,\ h\,(p_m)$

We then add rules that guess the value of P:

$pos(domain(P)) \leftarrow exec(domain(P)),\ not\ neg(domain(P))$, and

$neg(domain(P)) \leftarrow exec(domain(P)),\ not\ pos(domain(P))$.

The guess is then verified by the Hybrid ASP solver in the following way. In every state where $exec(domain(P))$ atom is present, P is evaluated. If the state contains atom $pos(domain(P))$ and the value of P is false, or if the state contains atom $neg(domain(P))$ and the value of P is positive then the state is rejected.

6 Conclusion

In this paper we introduced an action language Hybrid $\mathcal{ALE}$ in order to facilitate the development of diagnostic programs for the industrial data processing pipelines. The nature of the application is such as to require the diagnostic program to access outside sources. As precomputing all of the facts derivable from the outside sources can be impractical, access to those sources has to be done during inference. This poses a challenge to action languages that compile to ASP, since ASP does not provide mechanisms for accessing outside sources. We have thus chosen as a starting point action language Hybrid $\mathcal{AL}$, which compiles into Hybrid ASP—one of the extensions of ASP that provides access to outside sources. While Hybrid $\mathcal{AL}$ provides the syntactic structure for reasoning about the consequences of actions, it lacks structure such as found in C_{TAID} for reasoning about the actions themselves. We thus extended Hybrid $\mathcal{AL}$ with the structure for reasoning about actions, as found in C_{TAID}. The resulting action language Hybrid $\mathcal{ALE}$ can be viewed as a more expressive version of C_{TAID}

that compiles to Hybrid ASP instead of ASP. A system implementing Hybrid $\mathcal{AL}E$ was developed and is now being used at Google Inc. to help engineers with diagnosing malfunctions of certain data processing pipelines.

References

1. Balduccini, M., Gelfond, M.: Diagnostic reasoning with a-prolog. TPLP **3**(4-5) (2003) 425–461
2. Baral, C., Chancellor, K., Nam, T.H., Tran, N., Joy, A.M., Berens, M.E.: A knowledge based approach for representing and reasoning about signaling networks. In: Proceedings Twelfth International Conference on Intelligent Systems for Molecular Biology/Third European Conference on Computational Biology 2004, Glasgow, UK, July 31-August 4, 2004. (2004) 15–22
3. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In: Logic Based Artificial Intelligence, Kluwer Academic Publishers. (2000) 257–279
4. 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
5. Brik, A., Remmel, J.B.: Computing a Finite Horizon Optimal Strategy Using Hybrid ASP. In: NMR. (2012)
6. Brik, A., Remmel, J.B.: Action language hybrid AL. In Balduccini, M., Janhunen, T., eds.: Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings. Volume 10377 of Lecture Notes in Computer Science., Springer (2017) 322–335
7. 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
8. Dworschak, S., Grell, S., Nikiforova, V.J., Schaub, T., Selbig, J.: Modeling biological networks by action languages via answer set programming. Constraints **13**(1-2) (2008) 21–65
9. 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
10. 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
11. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. (1988) 1070–1080
12. Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. Artif. Intell. **2** (1998) 193–210
13. Lifschitz, V.: Answer set programming and plan generation. Artif. Intell. **138**(1-2) (2002) 39–54
14. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **8**(2) (2008) 129–165