Diagnosing Data Pipeline Failures Using Action Languages: A Progress Report

Alex Brik[1][0000−0002−0891−4355] and Jeffrey Xu[2][0000−0002−8159−073X]

[1] Google Inc., Mountain View, USA
[2] University of California Los Angeles

Abstract. In this paper we describe our work towards automating diagnosing failures of data processing pipelines at Google Inc. using action language Hybrid $\mathcal{ALE}$. We describe Diagnostic Modeling Library - a component providing a novel abstraction layer on top of Hybrid $\mathcal{ALE}$, describe the requirements and give an overview of our system, which has been deployed on a limited number of data processing pipelines.

Data processing pipelines (pipelines, for short) are software systems that process collections of data and produce either transformed data, aggregated data or some other output. Industrial pipelines can consist 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 stop data processing tasks, causing a set of cascading failures. The failures can cause an alert being dispatched to on-call engineers (on-calls, for short).

For the on-calls, an alert is a diagnostic challenge, as it can point to one of the later, rather than an earlier among the cascading failures. The earlier failures have to be found before the underlying problem can be resolved. Moreover, multiple possible causes of failure may have to be investigated. Automating the diagnosing process can decrease the time required to fix failures, and thus, improve the fault tolerance of the system and increase on-calls productivity.

Action languages [5] allow to formalize reasoning about effects of actions in dynamic domains. 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 [3] for an overview. In [1] an action language Hybrid $\mathcal{ALE}$ was introduced in order to facilitate the development of diagnostic programs for the industrial pipelines. Hybrid $\mathcal{ALE}$ provides a mechanism for accessing outside data sources with user provided algorithms. This feature of Hybrid $\mathcal{ALE}$ allows Hybrid $\mathcal{ALE}$ programs to gather information about pipelines from the outside sources in order to provide an accurate diagnosis. Unlike most other action languages that translate to ASP [4], Hybrid $\mathcal{ALE}$ translates to Hybrid ASP (H-ASP) [2] - an extensions of ASP that allows ASP-like rules to interact with outside sources.

In this paper we describe our work at Google Inc. on automating diagnosing of pipeline failures using Hybrid $\mathcal{ALE}$. Our diagnostic system is deployed on a limited number of pipelines. We start by specifying the requirements and by motivating the use of action language based approach in general and Hybrid $\mathcal{ALE}$ in particular. We then review Hybrid $\mathcal{ALE}$, and introduce Diagnostic Modeling Library - a component providing an abstraction layer on top of Hybrid $\mathcal{ALE}$. The use of Diagnostic Modeling Library simplifies model creation. We then discuss generating explanations and suggestion and give an overview of our system.

1 Requirements

Once on-calls receives a notification of a pipeline's failure, they typically have to perform the following tasks:

1. Determine the (most likely) causes of failure.
2. Obtain a detailed description of the failure, including possibly error messages produced by the failed job and other relevant information.
3. Understand why the failure has occurred.
4. Determine how to repair the failure.
5. Proceed with the repair.

Determining the causes of failure is only the first step in the repair process. An ideal system for helping on-calls would automate steps 1-5. In our work we focus on automating steps 1-4.

On-calls often operate at the description level of individual jobs, jobs' inputs and outputs. This is the lowest description level where failure and success can be quickly and effectively determined. This fact determines the types of models we focus on. Such a model can be represented as an acyclic digraph (dependency graph), where vertices are jobs and an edge from A to B represents the fact that some of the outputs of A are inputs of B. For such a model, the process of diagnosing the source of failure consists of starting with vertices without in-edges, and performing a breadth first search to determine the earliest set of vertices whose corresponding jobs have failed.

We now formulate the following engineering requirements for our system:

1. Provide a mechanism facilitating efficient creation of diagnostic models capable of the following: identifying pipeline jobs, describing dependencies between jobs, describing jobs' termination status.
2. Provide a mechanism for determining termination status of individual jobs (based on the external data sources)
3. Provide a mechanism for describing multiple possible diagnoses
4. Provide a mechanism for explaining the diagnoses in a user understandable form (possibly by gathering information from external data sources)
5. Provide a mechanism for generating suggestions for repairing the failures (possibly by gathering information from external data sources)

These requirements don't necessitate an action language based approach. We could create a library for describing dependency graphs that would model the pipelines. We could then provide mechanisms for association callbacks with graph vertices to determine the termination status of the corresponding jobs based on the external data sources. Our library would use the dependency graph to identify sources of failure. We could provide additional mechanisms for associating callbacks to generate explanations and suggestions. In the cases when an uncertainty exists, multiple trajectories would be examined.

There are two main reasons for choosing an action language based approach:

1. Availability of necessary functionality.
   (a) Answer set semantics provides a convenient mechanism for reasoning about multiple trajectories.
   (b) Action languages provide an elegant formalism for describing actions and their consequences, thus facilitating model creation.
2. Extensibility.
   (a) Typically, requirements of the software systems change over time. Because of that, a diagnostic system has to be easily extensible. Using formal languages as the basis for creating diagnostic models facilitates extensibility vs. an ad-hoc system.

Hybrid $\mathcal{ALE}$ provides an additional convenience: a principled way to combine arbitrary algorithms with the ASP-like rules, and a principled way to pass arbitrary data between such algorithms.

2 Hybrid $\mathcal{ALE}$

We now review 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 $\mathcal{AL}E$, 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 called *domain parameters* and time. We let $\mathbf{q}|_{domain}$ denote a vector of domain parameters only. 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 $\mathbf{p}|_{domain} = \mathbf{r}|_{domain}$. We note that there is a simple bijection between the set of transition diagrams and the set of hybrid transition diagrams.

In Hybrid $\mathcal{AL}E$, 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 $\neg p$. 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 $\mathcal{AL}E$ 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 $\mathcal{AL}E$ statements.

We omit the definition of semantics of Hybrid $\mathcal{ALE}$ for brevity. Interested readers are encouraged to consult [1].

3 Diagnostic Modeling Library

While creating diagnostic models using Hybrid $\mathcal{ALE}$ we have noticed repeated modeling patterns. In addition, requiring engineers to learn Hybrid $\mathcal{ALE}$ restricts the adoption of our diagnostic system. The Diagnostic Modeling Library encapsulates several modeling patterns expressed in Hybrid $\mathcal{ALE}$, and provides a convenient interface requiring a minimal understanding of action languages that focuses on a job as a basic modeling unit.

job := DeclareJob(job_name, job_prereqs)*:* declares a job *job_name*. *job_prereqs* specifies job's dependencies. The function specifies a single action *do(job_name)*, which is triggered if the prerequisites *job_prereqs* are satisfied. In order to add failure modes, *AllowFailure* or *TriggerFailure* functions need to be used in addition to *DeclareJob*. It's Hybrid $\mathcal{ALE}$ translation is:
default fluent finished_default(job_name)
do(job_name) causes finished_default(job_name)
finished(job_name) if finished_default(job_name)
trigger do(job_name) if job_prereqs, -finished(job_name)
succeeded(job_name) if finished_default(job_name), -failed(job_name)

job.AllowFailure(failure_type, failure_prereqs, FailureCheck, FailureCallback): specifies that a failure of type *failure_type* for the job can occur if *failure_prereqs* are satisfied and *FailureCheck* domain algorithm returns true. If the failure occurs, then the parameters in the consequent state are partly determined by *FailureCallback* action algorithm. The Hybrid $\mathcal{ALE}$ translation is:
allow failure_type(job.job_name) if job.job_prereqs, failure_prereqs,
-finished(job.job_name): FailureCheck
failure_type(job.job_name) causes failure(job.job_name) with FailureCallback

job.TriggerFailure(failure_type, failure_prereqs, FailureCheck, FailureCallback): specifies that a failure of type *failure_type* for the job is triggered if *failure_prereqs* are satisfied and *FailureCheck* domain algorithm returns true. If the failure occurs, then the parameters in the consequent state are partly determined by *FailureCallback* action algorithm. The Hybrid $\mathcal{ALE}$ translation is:
trigger failure_type(job.job_name) if job.job_prereqs, failure_prereqs,
-finished(job.job_name): FailureCheck
failure_type(job.job_name) causes failure(job.job_name) with FailureCallback

job.ValidateSuccess(invalidation_prereqs, InvalidationAlg): allows to invalidate a trajectory in case of job's success. In particular, if the job succeeds and *invalidation_prereqs* are satisfied and *InvalidationAlg* - a domain algorithm returns true, the trajectory becomes invalid. The Hybrid $\mathcal{ALE}$ translation is:
FALSE if finished_default(job.job_name), succeeded(job.job_name),
invalidation_prereqs: InvalidationAlg
Here *FALSE* is a special fluent that results in the trajectory becoming invalid.

job.ValidateFailure(invalidation_prereqs, InvalidationAlg): allows to invalidate a trajectory in case of job's failure. In particular, if the job fails and *invalidation_prereqs* are satisfied and *InvalidationAlg* - a domain algorithm returns true, the trajectory becomes invalid. The Hybrid $\mathcal{ALE}$ translation is:

FALSE if finished_default(job.job_name), failed(job.job_name),
    invalidation_prereqs: InvalidationAlg

This is an incomplete interface. Nevertheless, the five functions above are the most commonly used.

As an example the library usage, let's suppose that our pipeline consists of three jobs: A, B and C. Job B is dependent on job A, and job C is dependent on job B. Suppose that it is possible to determine whether job A has failed by using *AFailureCheck* algorithm, and it is possible to determine whether job C failed by using *CFailureCheck* algorithm. It is not possible to determine whether job B failed by any readily available algorithm. Nevertheless, we know that if job C succeeds, then job B must have succeeded. We can capture this information in the following model, with lines started with '#' indicating comments, and symbol '_' indicating an empty argument:

```
# Declare job A with no prerequisites
jobA := DeclareJob(A, _)
# Specify that AFailureCheck identifies A's failure
jobA.TriggerFailure(failure, _, AFailureCheck, _)
# Declare job B dependent on job A
jobB := DeclareJob(B, finished(A))
# Specify that job B can fail
jobB.AllowFailure(failure, _, _, _)
# Declare job C dependent on job B
jobC := DeclareJob(C, finished(B))
# Specify that CFailureCheck identifies C's failure
jobC.TriggerFailure(failure, _, CFailureCheck, _)
# Specify that job C's success invalidates job B's failure
jobC.ValidateSuccess(failed(B), _)
```

4 Generating Explanations and Suggestions

Repairing the failures of pipelines can be facilitated if the diagnostic software provides explanations or other relevant information about the source of the failure, and if the diagnostic software provides the suggestions for repairing.

Both the explanations and the suggestions can be specific to a trajectory and to a failure. We did not attempt to solve the problem of automatic generation of explanation and suggestions based on the formal model of a diagnosed system. Nevertheless, we have made some initial steps in providing useful additional information about the failures and possible ways to repair them to on-calls.

Our approach uses a combination of user generated information and information from outside sources about the failure. Often, engineers familiar with a pipeline can provide a short list of failure descriptions and another list with suggestions for repairing. These can be integrated with the diagnostic model with the help of lookup tables. In the tables, both explanations and suggestions are keyed by an action representing a failure or by a specific fluent generated during a failure. Messages in the tables can be automatically customized based on the date or other parameters specific to the particular diagnosing evaluation, thus making them more helpful to on-calls.

Additional information about the failures can be retrieved from the outside data sources using action algorithms. In the modeling diagnostic library, both *AllowFailure* and *TriggerFailure* functions can be evoked with *FailureCallback* action algorithm. *FailureCallback* algorithm can retrieve information, such as error messages generated by the failed job and use it to generate an explanation recorded in *explanation(job_name)* parameter in the consequent state. When a diagnosis is reported, the value of *explanation(job_name)* parameter is reported as well. A similar mechanism can also be used to generate suggestions based on the information from the outside sources. Such an explanation or a suggestion is reported together with an associated action or a fluent, thus providing a more meaningful description of the failure.

5 Job Termination Status and Automatic Model Generation

Data processing pipelines at Google are typically run using Borg system [6]. This provides several advantages. First, data processing pipelines are described using BCL configuration files [6]. A BCL description of a pipeline contains a description of all the jobs in the pipeline as well as the dependencies. Second, in many cases, it is possible to determine whether a particular job run failed or succeeded using the Borg monitoring system, and in some cases to retrieve error messages generated by the failed jobs as well as other relevant information.

We have used these features of the Borg system for the following:

1. To facilitate automatic creation of basic diagnostic models using the Model Diagnostic Library
2. To automatically determine job's run's termination status as well to automatically generate failure explanations by retrieving error messages generated during job's run
3. When the automatically generated model is insufficient, it can serve as a skeleton for a more detailed manually enhanced model.

This architecture is illustrated in figure 1. Pipeline configuration is used by model generator to create a model. The automatically created model contains callbacks that access data relevant for determining jobs' termination status. The model can then be refined with the help of the engineers familiar with the pipeline. The model is expressed using diagnostic library. Diagnostic library translates the model into a Hybrid $\mathcal{ALE}$ description, and Hybrid $\mathcal{ALE}$ description is further translated into Hybrid ASP. Hybrid ASP solver then uses the model to generate diagnoses, and possibly explanations and suggestions.

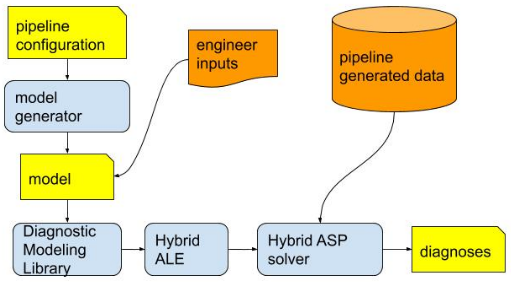

Fig. 1. Diagnostic system architecture.

6 Conclusion

In this paper we discussed our work on a system deployed on a limited number of pipelines at Google Inc. for automating diagnosing of pipeline failures using action language Hybrid $\mathcal{ALE}$. We specified the requirements that guided our work, and motivated the use of Hybrid $\mathcal{ALE}$. To simplify model creation we introduced Diagnostic Modeling Library - a component providing an abstraction layer on top of Hybrid $\mathcal{ALE}$. We discussed initial progress in extending the functionality of a diagnostic system to include explanation and suggestion generation - the functionality that makes a diagnostic system more useful for on-calls. We reviewed the architecture of our system: from automatic model generation, and manual model refinement to generating diagnosis based on the model and external data. Our approach is generally applicable and is not Google specific.

References

1. Bomanson, J., Brik, A.: Diagnosing data pipeline failures using action languages. In Balduccini, M., Lierler, Y., Woltran, S., eds.: Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings. Volume 11481 of Lecture Notes in Computer Science., Springer (2019) 181–194
2. 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
3. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
4. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. (1988) 1070–1080
5. Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. Artif. Intell. **2** (1998) 193–210
6. 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)