Diagnosing Automatic Whitelisting for Dynamic Remarketing Ads Using Hybrid ASP

Alex Brik[1] and Jeffrey Remmel[2]

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

Abstract. Hybrid ASP (H-ASP) is an extension of ASP that allows users to combine ASP type rules and numerical algorithms. Dynamic Remarketing Ads is Google's platform for serving customized ads based on past interactions with a user. In this paper we will describe the use of H-ASP to diagnose failures of the automatic whitelisting system for Dynamic Remarketing Ads. We will show that the diagnosing task is an instance of a computational pattern that we call the Branching Computational Pattern (BCP). We will then describe a Python H-ASP library (H-ASP PL) that allows to perform computations using a BCP, and we will describe a H-ASP PL program that solves the diagnosing problem.

Past research has demonstrated that logic programming with the answer-set semantics, known as *answer-set programming* or *ASP*, for short, is an expressive knowledge-representation formalism [2, 14, 16, 21, 20, 22]. The availability of the non-classical negation operator *not* allows the user to model incomplete information, frame axioms, and default assumptions such as normality assumptions and the closed-world assumption efficiently. Modeling these concepts in classical propositional logic is less direct [14] and typically requires much larger representations.

A fundamental methodological principle behind ASP, which was identified in [21], is that to model a problem, one designs a program so that its answer sets *encode* or *represent* problem solutions. Niemelä [22] has argued that logic programming with the stable-model semantics should be thought of as a language for representing constraint satisfaction problems. Thought of from this point of view, ASP systems are ideal logic-based systems to reason about a variety of types of data and integrate quantitative and qualitative reasoning. ASP systems allow the users to describe solutions by giving a series of constraints and letting an ASP solver such as *cmodels* [15], *smodels* [24], *assat* [17], *clasp* [13], and *dlv* [8], *pbmodels* [18] or *aspps* [10, 9] search for solutions. Such systems can, in principle, solve any NP-search problem [19].

To solve many of the real world problems one needs to have an ASP programming environment where one can perform external data searches and bring back information that can be used in the program. Extensions of ASP that allow such external data searches include DLV^{DB} system [25] for querying relational databases, VI programs [7] for importing knowledge from external sources, HEX programs [11] which allow access to external data sources via external atoms, $GRINGO$ grounder that provides an interface for calling function written in Lua during the grounding process [12], and Hybrid ASP (H-ASP) introduced by the authors in [5].

In this paper, we will discuss applications of H-ASP which can perform external data searches. In particular, we shall discuss an example of problems that can be solved by processing a connected directed acyclic graph (cDAG for short) where each vertex of the cDAG contains both logical and non-logical information in the form of parameters. The cDAG can be generated by following a multi-step pattern of computation which we will call the Branching Computational Pattern (BCP

for short). At any stage in the computation we are given a set of vertices. These vertices can either be an initial set of vertices or a set of vertices produced at the previous step. Then the BCP instance creates multiple new branches emanating from a particular vertex. For each new branch, the BCP instance performs a computation using the data from the vertex and possibly auxiliary data from external repositories to derive new logical information and parameters at that vertex as well as pass relevant logical information and new parameters to its children. The result of such computation defines new vertices of the cDAG. Then new edges from each parent vertex to its children are added to the cDAG. The resulting cDAG is called a BCP cDAG.

The focus of this paper is the problem of diagnosing failures of the automatic whitelisting system for Dynamic Remarketing Ads (automatic whitelisting, for short). Dynamic Remarketing Ads is Google's platform for delivering ads which are customized to an individual user based on the user's past interactions with the advertiser such as the user's previously viewed items or abandoned shopping carts. In order for Google to start serving dynamic remarketing ads for a particular advertiser that advertiser needs to be whitelisted, i.e. the advertiser has to have been added to a list of advertisers that are known to use Dynamic Remarketing Ads. Whitelisting is done automatically by a system that detects whether an advertiser is ready to serve dynamic remarketing ads based on the logs and the content of ads databases.

There are nine cases when an advertiser can be automatically whitelisted. In each case there is a set of constraints that need to be satisfied in order for that case to apply. The technical challenge in using ASP for diagnosing automatic whitelisting is that in order to check the constraints it is necessary to search data stored in Google's various data repositories. The fact that in our application, the amount of data is quite large and the repository contents change in real time makes pre-computing impractical. Moreover, data searches in repositories often depend on the data obtained in the previous steps. Under these circumstances what is required is an extension of ASP that allows the following: (1) conclusions to be derived conditional on the results of the external data searches, and (2) parameter passing between the algorithms that perform data searches. H-ASP provides this functionality. To solve the problem of diagnosing failures of the automatic whitelisting system we have implemented a Python library, which we call H-ASP PL for running H-ASP programs that use a certain subset of H-ASP rules. We have then created a H-ASP program that runs using H-ASP PL library. The program was successfully used for several months in the cases of many advertisers.

Another problem that can be solved by processing a cDAG is computing an optimal strategy for an agent acting in a dynamic domain. In [6], the authors showed how H-ASP programs can be used to combine logical reasoning, continuous parameters, and probabilistic reasoning in the context of computing optimal strategy using Markov Decision Processes. A feature of the solution in [6] was that one started with a basic H-ASP program and performed a series of program transformations so that one could compute a maximal stable model which contains all of the stable models of the original program. That is, according to the H-ASP semantics, which we will define in a subsequent section, a H-ASP program can have multiple stable models where some of the stable models can be subsets of other stable models. In the context of computing an optimal strategy, such stable models would describe only a part of the evolution tree of the dynamic domain. Such dynamic domains can be represented as BCP cDAGs. Hence, each stable model represents a part of the BCP cDAG. However, in order to compute an optimal strategy, one needs to compute the entire evolution tree of a dynamic domain. Thus the full BCP cDAG is required.

We have a similar situation in the case of diagnosing automatic whitelisting. We are also interested in the full BCP cDAG which in our case will encode all of the possible whitelisting paths.

There already exists a literature discussing the use of ASP for diagnosing malfunctioning devices. In [1] Balduccini and Gelfond describe an approach for diagnosing a malfunctioning device based on the theory of the action language $\mathcal{AL}$. In their approach the underlying diagnostic program explicitly describes the laws that govern the behavior of the dynamic domain, and the non-monotonicity of ASP is used to compute all the possible scenarios under which a malfunction could occur. In the case of diagnosing automatic whitelisting, we use the non-monotonicity of ASP to compute all the possible scenarios for a malfunction, however we do not explicitly describe the laws that govern the behavior of the dynamic domain. The latter is motivated by the relative simplicity of the domain for the automatic whitelisting and by the time constraints of the project.

There are two main advantages of using H-ASP rather than a common programming language such as Python directly. The first advantage is the efficiency of representation. Our H-ASP program specifies how automatic whitelisting occurs and lets the solver report back failures when automatic whitelisting fails. An equivalent Python program will have to specify both how the automatic whitelisting occurs and the details of the diagnostic logic. Hence, H-ASP program is smaller than one would expect the equivalent Python program to be. The second advantage is the robustness of the H-ASP program. Because H-ASP program describes mostly the problem domain, it is easier to update the program when changes to the automatic whitelisting logic occur. This is important since the requirements for automatic whitelisting are continually being modified.

The outline of this paper is the following. In Section 1, we formally define the Branching Computational Pattern (BCP). Our problem of diagnosing automatic whitelisting is a special case of the BCP. In Section 2, we give an overview of H-ASP. In Section 3, we discuss the computational pattern as it relates to H-ASP and we briefly describe the H-ASP PL library. In Section 4, we present a toy example to illustrate how the problem of diagnosing automatic whitelisting is solved. In Section 5, we describe the semantics of H-ASP PL, and the Local Algorithm which is used in H-ASP PL. In Section 6, we discuss some of the related work and give the closing comments.

1 The Branching Computational Pattern (BCP)

Let $\langle V, E\rangle$ be a connected directed acyclic graph (cDAG). Let $R(V, E)$ be the set of vertices with no in-edges. If $|R(V, E)| = 1$, then we will refer to the unique vertex $r(V, E) \in R(V, E)$ as the root node. If $(v, w) \in E$, we will say that v is a *parent* of w and that w is a *child* of v. If there exists $v_1, v_2, ..., v_n$ such that for all $i \in \{2, ..., n\}$, $(v_{i-1}, v_i) \in E$ and $v_1 = v$ and $v_n = w$, then we say that v is an *ancestor* of w and w is a *descendant* of v. It is easy to see that for all $v \in V$ such that $v \notin R(V, E)$, there exists a vertex in $R(V, E)$ which is an ancestor of v.

Our branching computational patterns allow the user to compute a cDAG $\langle V, E\rangle$ where each vertex $v \in V$ is a pair $(A, \mathbf{p})$ where A is a set of propositional atoms and $\mathbf{p}$ is a vector of parameter values representable by a computer. We will refer to such a cDAG as a *computational cDAG*. If all $(A, \mathbf{p}) \in V$, $A \subseteq At$ and $\mathbf{p} \in S$, then we will say that $\langle V, E\rangle$ is a *computational cDAG over At and S*. At each cDAG vertex $(A, \mathbf{p})$, the computation consists of the two steps:
1. use A and $\mathbf{p}$ to choose algorithms (that will possibly access external data repositories and/or perform computations) to produce the set of next parameter value vectors $\mathbf{q}_1, ..., \mathbf{q}_k$ and
2. for each $\mathbf{q}_i$ produced in step 1, derive atoms $B_{i,1}$, $B_{i,2}$, ..., B_{i,m_i}.

The set of children of $(A, \mathbf{p})$ will be the pairs $(\{B_{i,1}, B_{i,2}, \ldots, B_{i,m_i}\}, \mathbf{q}_i)$ for $i = 1, \ldots, k$. To produce the root nodes of the cDAGs, step 2 is applied to the initial set of parameter values specified as an input. The computation can then be repeated at each child node.

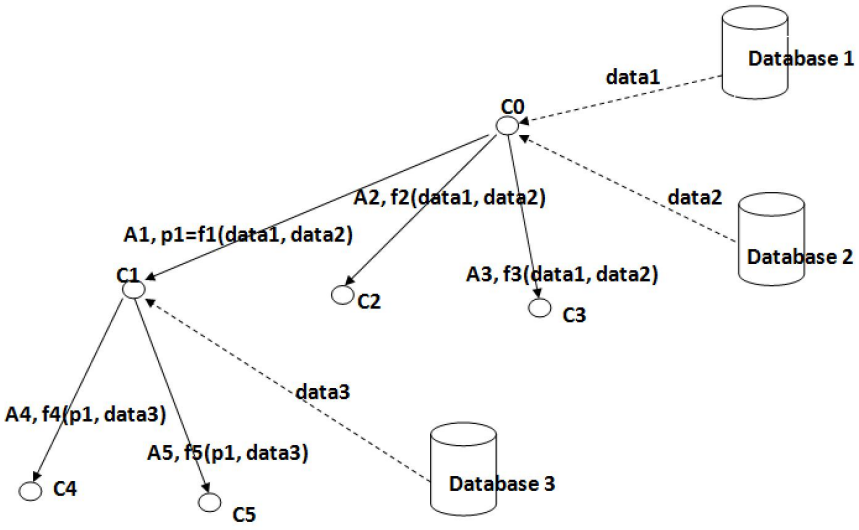

Fig. 1. Computational pattern illustration

This computational process is illustrated in Fig. 1. At the node *C0*, we obtain *data1* from database *Database1* and *data2* from database *Database2*. *C0* then creates three new children: $(\{A_i\}, fi(data1, data2))$ for $i = 1, 2, 3$ where fi is one of the computational algorithms associated with an H-ASP rule that can be applied at *C0*. At node *C1*, we obtain *data3* from database *Database3*. Then *C1*, creates two new children: $(\{A_i\}, fi(p1, data3))$ for $i = 4, 5$. Fig. 1 illustrates the main aspects of the BCP. At each node, external data sources can be accessed. This new data, the value of the parameters stored at the node, and the logical information stored at the node are then used to create new children by passing new parameters and logical information to each child as well as updating the logic information stored at the node.

The problem of diagnosing failures for automatic whitelisting fits the general BCP paradigm. There are nine cases for the automatic whitelisting. In order to help in understanding the types of criteria used, we shall describe one of these cases.

Case one: An advertiser is whitelisted if the advertiser has installed a Javascript tag containing the id of one of the advertiser's products, a user visits advertiser's website, and the advertiser has created a dynamic remarketing ad.

Here the initial set of candidate advertisers for whitelisting is obtained from the table, which we will call T_init, containing the information about advertisers who have installed a Javascript tag. Whether a user has visited advertiser's website can be determined by examining a log called L_userevents. The id of each advertiser from the candidate set will be used to search L_userevents to determine whether a user has visited advertiser's website or not. The remaining advertiser ids will be used to determine whether the advertiser has created a dynamic remarketing ad by using an external function GetCreatedAdIds().

The cDAG representing the automatic whitelisting system can be constructed by making a vertex represent a whitelisting condition that needs to be satisfied. An edge from a vertex x to a vertex y will be added to indicate that the condition for y needs to be checked immediately after

checking the condition for x . This may be necessary, for instance if data derived when checking the condition for x needs to be used to check the condition for y. A root node will have as its children the first conditions for each of the nine cases. Thus the cDAG will be a tree with 9 branches.

2 Hybrid ASP

In this section we shall give a brief overview of H-ASP. A H-ASP program P has an underlying parameter space S and a set of atoms At. Elements of S are of the form $\mathbf{p} = (t, x_1, \ldots, x_m)$ where t is time and x_i are parameter values. We shall let $t(\mathbf{p})$ denote t and $x_i(\mathbf{p})$ denote x_i for $i = 1, \ldots, m$. We refer to the elements of S as *generalized positions*. The universe of P is $At \times S$. For ease of notation, we will often identify an atom and the string representing an atom.

Let $M \subseteq At \times S$. Define $\widehat{M} = \{\mathbf{p} \in S : (\exists a \in At)((a, \mathbf{p}) \in M)\}$. For a generalized position $\mathbf{p} \in S$, define $W_M(\mathbf{p}) = \{a \in At : (a, \mathbf{p}) \in M\}$. A *hybrid state* at generalized position $\mathbf{p} \in S$ is a pair $(W_M(\mathbf{p}), \mathbf{p})$. In general, a pair $(A, \mathbf{p})$ where $A \subseteq At$ and $\mathbf{p} \in S$ will be referred to as a *hybrid state*. For a hybrid state $(A, \mathbf{p})$, we write $(A, \mathbf{p}) \in M$ if $\mathbf{p} \in \widehat{M}$ and $W_M(\mathbf{p}) = A$.

A *block* B is an object of the form $B = a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_k$ where $a_1, \ldots, a_n, b_1, \ldots, b_k \in At$. We let $B^- = not\ b_1, \ldots, not\ b_k$. Given $M \subseteq At \times S$, $B = a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_k$, and $\mathbf{p} \in S$, we say that M satisfies B at the generalized position $\mathbf{p}$, written $M \models (B, \mathbf{p})$, if $(a_i, \mathbf{p}) \in M$ for $i = 1, \ldots, n$ and $(b_j, \mathbf{p}) \notin M$ for $j = 1, \ldots, k$. If B is empty, then $M \models (B, \mathbf{p})$ automatically holds.

There are two types of rules in H-ASP.

Advancing rules are of the form $a \leftarrow B_1; B_2; \ldots; B_r : A, O$
where A is an algorithm, each B_i is a block, and $O \subseteq S^r$ is such that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \ldots < t(\mathbf{p}_r)$, $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \subseteq S$, and for all $\mathbf{q} \in A(\mathbf{p}_1, \ldots, \mathbf{p}_r)$, $t(\mathbf{q}) > t(\mathbf{p}_r)$. Here and in the next rule, we allow n or k to be equal to 0 for any given i. Moreover, if $n = k = 0$, then B_i is empty and we automatically assume that B_i is satisfied by any $M \subseteq At \times S$. We shall refer to O as the *constraint set* of the rule and the algorithm A as the *advancing algorithm* of the rule. The idea is that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$ and for each i, B_i is satisfied at the generalized position $\mathbf{p}_i$, then the algorithm A can be applied to $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ to produce a set of generalized positions O' such that if $\mathbf{q} \in O'$, then $t(\mathbf{q}) > t(\mathbf{p}_r)$ and $(a, \mathbf{q})$ holds.

Stationary rules are of the form $a \leftarrow B_1; B_2; \ldots; B_r : H, O$
where each B_i is a block, $O \subseteq S^r$ is such that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \cdots < t(\mathbf{p}_r)$, and H is a Boolean algorithm defined on O. We shall refer to O as the *constraint set* of the rule and the algorithm H as the *Boolean algorithm* of the rule. The idea is that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$ and for each i, B_i is satisfied at the generalized position $\mathbf{p}_i$, and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ is true, then $(a, \mathbf{p}_r)$ holds.

A *H-ASP Horn program* is a H-ASP program which does not contain any negated atoms in At. Let P be a Horn H-ASP program, let $I \in S$ be an initial condition. Then the one-step provability operator $T_{P,I}$ is defined so that given $M \subseteq At \times S$, $T_{P,I}(M)$ consists of M together with the set of all $(a, J) \in At \times S$ such that
(1) there exists a stationary rule $C = a \leftarrow B_1; B_2; \ldots; B_r : H, O$ and $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $(a, J) = (a, \mathbf{p}_r)$, $M \models (B_i, \mathbf{p}_i)$ for $i = 1, \ldots, r$, and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 1$ or
(2) there exists an advancing rule $C = a \leftarrow B_1; B_2; \ldots; B_r : A, O$ and $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $J \in A(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ and $M \models (B_i, \mathbf{p}_i)$ for $i = 1, \ldots, r$.

The stable model semantics for H-ASP programs is defined as follows. Let $M \subseteq At \times S$ and $I \in S$. An H-ASP rule $C = a \leftarrow B_1; \ldots, B_r : A, O$ is *inconsistent* with (M, I) if for all $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$, either (i) there is an i such that $M \not\models (B_i^-, \mathbf{p}_i)$, (ii) $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M} = \emptyset$ if A is an advancing algorithm, or (iii) $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 0$ if A is a Boolean algorithm. Then 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 $C = a \leftarrow B_1; \ldots, B_r : A, O$ is not eliminated by (1), then replace it by $a \leftarrow B_1^+; \ldots, B_r^+ : A^+, O^+$ where for each i, B_i^+ is the result of removing all the negated atoms from B_i, O^+ is equal to the set of all $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ in $O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $M \models (B_i^-, \mathbf{p}_i)$ for $i = 1, \ldots, r$ and $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M} \neq \emptyset$, and $A^+(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ is defined to be $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M}$.
(3) If the stationary rule $C = a \leftarrow B_1; \ldots, B_r : H, O$ is not eliminated by (1), then replace it by $a \leftarrow B_1^+; \ldots, B_r^+ : H|_{O^+}, O^+$ where for each i, B_i^+ is the result of removing all the negated atoms from B_i, O^+ is equal to the set of all $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ in $O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $M \models (B_i^-, \mathbf{p}_i)$ for $i = 1, \ldots, r$ and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 1$.

Then M is a *stable model of P with initial condition I* if $\bigcup_{k=0}^{\infty} T_{P^{M,I},I}^k(\emptyset) = M$.

We say that M is a *single trajectory stable model* of P with initial condition I if M is a stable model of P with initial condition I and for each $t \in \{t(\mathbf{p}) | \mathbf{p} \in S\}$, there exists at most one $\mathbf{p} \in \widehat{M} \cup \{I\}$ such that $t(\mathbf{p}) = t$.

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. We say that an advancing algorithm A *fixes* a parameter y if A does not let y be free.

The reason for introducing the last two definitions is that we often want to limit the effects of algorithms to specifying only a subsets of the parameters to make programs easier to understand. The exact mechanism for doing so will be discussed in the next section. For now, however we will note that the parameters that the advancing algorithm will be responsible for producing will correspond to the fixed parameters of the algorithm. The rest of the parameters will correspond to the free parameters.

3 H-ASP Library

H-ASP programs can be used to perform the computations for a BCP. In fact, BCP computations can be carried by H-ASP programs which use only a restricted set of H-ASP rules which we call H-ASP programs of order 1. A H-ASP program P is of order 1 if all its advancing rules are of the form $a \leftarrow B : A, O$, and all its stationary rules of the form $a \leftarrow B : H, O$. If P is of order 1, then we will say that $\mathbf{p}$ is a *child* of $\mathbf{q}$ (and $\mathbf{q}$ is a parent of $\mathbf{p}$) under P, I if there exists a stable model M of P with the initial condition I and there exists an advancing rule $a \leftarrow B : A, O \in P$ such that $M \models (B, \mathbf{q})$ and $\mathbf{q} \in O$ and $\mathbf{p} \in A(\mathbf{q})$.

Given an H-ASP program P of order 1 and the initial condition I, we define the computational cDAG *induced* by P, I, $comp(P, I) = \langle V, E \rangle$, as follows.
(1) V is the set of all the hybrid states $(A, \mathbf{p})$ such that there exists a stable model M of P with initial condition I and $(A, \mathbf{p}) \in M$.

(2) E is the set of all pairs $((A,\mathbf{p}),(B,\mathbf{q}))\in V^2$ such that there exists a stable model M of P initial condition I and $(A,\mathbf{p})\in M$ and $(B,\mathbf{q})\in M$ and $(A,\mathbf{p})$ is a parent of $(B,\mathbf{q})$ under P,I.

We can prove the following theorem.

Theorem 1. *Let At be a set of propositional atoms and let S be a set of parameter values. Let $\langle V,E\rangle$ be a computational cDAG over At and S. Then there exists a H-ASP program P of order 1 and an initial condition I for P such that $comp(P,I)=\langle W,U\rangle$ is isomorphic to $\langle V,E\rangle$. Moreover, P can be chosen to have a maximal stable model M and a parameter space X such that there exists a map $\pi:X\to S$ so that the isomorphism g from $comp(P,I)$ to $\langle V,E\rangle$ is defined by setting $g((A,\mathbf{p}))=(A\cap At,\pi(\mathbf{p}))$ for $(A,\mathbf{p})\in W$ where W is the set of all the hybrid states of M.*

The proof of the theorem consists of construction of a H-ASP program P of order 1, an initial condition I, and a simple isomorphism π that satisfy the conditions of the theorem. The idea is that for every edge $((A,\mathbf{p}),(B,\mathbf{q}))\in E$, P contains a set of rules with constraint sets that are satisfied only by $\pi(\mathbf{p})$, and that generate $(B\cap At,\pi(\mathbf{q}))$.

It is also easy to see that for a H-ASP program P of order 1 and an initial condition I, the computation of a stable model is performed according to the BCP along the computational cDAG induced by P,I.

Practical applications of BCP's require either a computer language or a library. Due to the time constraints of our project, we created a Python library which allows us to compute the stable models for H-ASP programs of order 1. However, to make the task of programming with the library easier, we have added one more type of rule beyond those allowed in H-ASP programs of order 1. We will now briefly describe some of the key features of the library and the programs called H-ASP PL programs that it processes.

Let At be the set of atoms and let S be the parameter space.

1. A H-ASP PL program consists of a collection of three types of H-ASP rules: advancing rules of the form $a\leftarrow B:A,O$, stationary rules of the form $a\leftarrow B:H,O$, and stationary rules of the form $a\leftarrow B_1;B_2:G,\Theta$.

2. The parameters in the parameter space S are named. If $\mathbf{p}$ is a generalized position and Q is a parameter, we denote the value of Q at $\mathbf{p}$ by $\mathbf{p}[Q]$.

3. The time parameter is named TIME and it is assumed that every advancing algorithm increments the value of TIME by 1. That is, if A is an advancing algorithm, $\mathbf{p}$ is a generalized position, then for all $\mathbf{q}\in A(\mathbf{p})$, $\mathbf{q}[\text{TIME}]=\mathbf{p}[\text{TIME}]+1$. Because of this assumption, the advancing algorithms are not required to specify the value of the parameter TIME.

4. In [5], the authors suggested an indirect approach by which the advancing algorithms can specify the values for only some of the parameters. The approach requires extending the Herbrand base of a program P by a set of new atoms S_1, ..., S_m one for each parameter. Suppose that there is an advancing algorithm A in a rule $a\leftarrow B:A,O$ that specifies parameters with indexes i_1, i_2, ..., i_k and lets other parameters be free. Then we add to P rules of the form $S_{i_j}\leftarrow B:A,O$ for each j from 1 to k. This is, repeated for every advancing rule of P. Then if M is a stable model of P and $\mathbf{p}\in\widehat{M}$, we will require that $\{S_1,\ldots,S_m\}\subseteq W_M(\mathbf{p})$. This will ensure that every parameter at $\mathbf{p}$ is set by some advancing rule. For our library, we assume that this mechanism is used by any H-ASP program that it will process. This allows us to implement it implicitly without requiring the H-ASP user to specify the additional rules.

5. For our application, we would like to have the ability to apply a constraint to two hybrid states belonging to the same single trajectory stable model of P. In order to do that in our library, for a stationary rule of the form $a\leftarrow B_1;B_2:G,\Theta$, we assume that if $G(\mathbf{p},\mathbf{q})=1$, then $t(\mathbf{p})+1=t(\mathbf{q})$

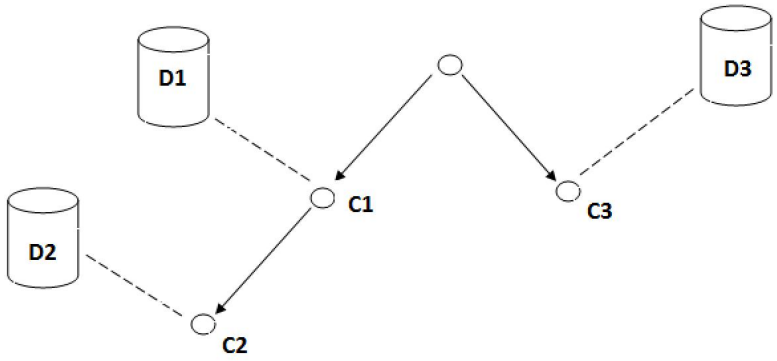

Fig. 2. Decision tree diagram

and there exists a single trajectory stable model M of P such that $\{\mathbf{p}, \mathbf{q}\} \subseteq \widehat{M}$. (The details can be found in [4].)

4 Example

The following example illustrates a typical step of processing performed by the program for diagnosing automatic whitelisting. The complete diagnosis requires many steps of this type.

Suppose that a decision is to be made based on a decision tree containing two branches. In the first branch data repository 1 is searched for the data D1 relevant for condition C1. If condition C1 is satisfied based on D1, then data repository 2 is searched for an additional data D2[D1] which is dependent on D1. Condition C2 is then evaluated based on D2[D1]. If the condition C2 is satisfied, then an affirmative decision is made. If either the condition C1 or the condition C2 are not satisfied, then the decision is made based on the evaluations for the second branch. In the second branch, data repository 3 is searched for the data D3. Condition C3 is applied to D3 and if the condition is satisfied, then an affirmative decision is made. If in neither branch 1 nor branch 2, an affirmative decision is made, then a negative decision is made (see Fig. 2).

Suppose that we would like to explain why a negative decision was made. An explanation would have to describe which condition in each of the two decision branches has failed.

We will use the following idea to create a H-ASP PL program P that will generate an explanation. For each of the conditions C1, C2, C3, the program P will produce a state that will contain all of the information necessary to make the decision for the corresponding Ci. Since to make the decision C2 it is necessary to use data that is required to make the decision C1, the state corresponding to C2 will be a child of the state corresponding to C1. The states corresponding to C3 and C1 will be children of the initial state. In effect, P will produce a stable model whose computational cDAG will model the decision tree in that the nodes of the computational cDAG will correspond to the conditions of the decision tree, the edges will correspond to the successor relations of the tree.

We will need the following H-ASP PL parameters: DATA - to pass the data from the state corresponding to C1 to the state corresponding to C2, EXPLANATION - to record the description of the conditions that were not satisfied. We will assume that the advancing algorithm SetData1

gets data D1 and sets the output parameter DATA. The advancing algorithm SetData2 uses the value of DATA parameter in order to get data D2[D1]. The algorithm then sets the parameter DATA of the produced generalized position D2[D1]. The advancing algorithm SetData3 gets data D3 and sets output parameter DATA.

In order to check condition Ci where $i \in \{1, 2, 3\}$, we will assume that the Boolean algorithms CheckCondition{i} for $i \in \{1, 2, 3\}$ are implemented and return TRUE iff Ci is satisfied.

EXPLANATION parameter will be set by the advancing algorithms Condition{i}Fails, which will set the value of EXPLANATION parameter to a string "condition {i} is not satisfied" for $i \in \{1, 2, 3\}$. The advancing algorithm SetExplanationEmpty sets the value of EXPLANATION to the empty string.

This gives us the following H-ASP PL rules (rule label is in the brackets in the following format R{branch#}{time that the rule will affect}.{rule index}):

```
# Initial branching. IsTime0 is a boolean algorithm that returns TRUE iff the
# time of the input generalized position is 0.
  [R1.0.1] BRANCH1 :- not BRANCH2: IsTime0
  [R2.0.1] BRANCH2 :- not BRANCH1: IsTime0
# Produce generalized position for making decision C1
  [R1.1.1] CHECK_C1 :- BRANCH1: SetData1
  [R1.1.2] CHECK_C1 :- BRANCH1: SetExplanationEmpty
# Check condition C1
  [R1.1.3] C1_SAT :- CHECK_C1: CheckCondition1
  [R1.1.4] C1_DONE :- CHECK_C1
# If condition C1 is not satisfied then generate the appropriate explanation
# and set DATA to empty - branch 1 negative decision is explained.
  [R1.2.1] EXPLAINED :- C1_DONE, not C1_SAT: Condition1Fails
  [R1.2.2] END :- C1_DONE, not C1_SAT: SetDataEmpty
# If condition C1 is satisfied, then proceed with getting D2[D1]
# Data D1 is the value of DATA for the appropriate generalized position
  [R1.2.3] CHECK_C2 :- C1_SAT: SetData2
  [R1.2.4] CHECK_C2 :- C1_SAT: SetExplanationEmpty
# Check condition C2
  [R1.2.5] C2_SAT :- CHECK_C2: CheckCondition2
  [R1.2.6] C2_DONE :- CHECK_C2
# If condition C2 is not satisfied then produce the corresponding explanation
  [R1.3.1] EXPLAINED :- C2_DONE, not C2_SAT: Condition2Fails
# If condition C2 is satisfied, set EXPLANATION to empty
  [R1.3.2] END :- C2_DONE, C2_SAT: SetExplanationEmpty
# In both cases set DATA to empty
  [R1.3.3] END :- C2_DONE: SetDataEmpty
# Now, for branch 2 - get D3
  [R2.1.1] CHECK_C3 :- BRANCH2: SetData3
  [R2.1.2] CHECK_C3 :- BRANCH2: SetExplanationEmpty
# Check condition C3
  [R2.1.3] C3_SAT :- CHECK_C3: CheckCondition3
  [R2.1.4] C3_DONE :- CHECK_C3
# If C3 is not satisfied then state that in EXPLANATION, otherwise set
```

```
# EXPLANATION to empty
  [R2.2.1] EXPLAINED :- C3_DONE, not C3_SAT: Condition3Fails
  [R2.2.2] END :- C3_SAT: SetExplanationEmpty
# In both cases set DATA to empty
  [R2.2.3] END :- C3_DONE: SetDataEmpty
```

We will now consider an example of a stable model that the above rules can produce. For the purpose of our example, suppose that the conditions C2 and C3 are not satisfied, whereas the condition C1 is satisfied. The stable model will be represented by a list of hybrid states $(A, \mathbf{p})$ where $\mathbf{p}$ is a generalized position and A is a set of atoms corresponding to $\mathbf{p}$. A generalized position will be represented as a tuple (t, D, E) there t is time, D is data and E is an explanation.

```
# Apply R1.0.1 and R2.0.1
  <{BRANCH1}, (0, null, null)>
  <{BRANCH2}, (0, null, null)>
# For the branch 1 time 1:
# R1.1.1 is used to get data D1, R1.1.2 sets EXPLANATION parameter to ""
# R1.1.3 will produce the atom C1_SAT since C1 is satisfed by assumption
# R1.1.4 will produce the atom C1_DONE
  <{CHECK_C1, C1_SAT, C1_DONE}, (1, D1, "")>
# For branch 1 time 2:
# The premises of R1.2.1 and R1.2.2 are not satisfied
# R1.2.3 will produce D2[D1] and R1.2.4 will set the parameter EXPLANATION to ""
# R1.2.5 will not produce anything since CheckCondition2 will return FALSE
# R1.2.6 will produce the atom C2_DONE
  <{CHECK_C2, C2_DONE}, (2, D2[D1], "")>
# For branch 1 time 3:
# R1.3.1 will produce the atom EXPLAINED and set the parameter EXPLANATION to
# "condition 2 is not satisfied"
# R1.3.2 will not produce anything since C2_SAT is not in the previous state
# R1.3.3 will produce the atom END and will set DATA=null
  <{EXPLAINED, END}, (3, null, "condition 2 is not satisfied")>
# For branch 2 time 1:
# R2.1.1 will produce the atom CHECK_C3 and will set the parameter DATA to D3
# R2.1.2 will set the parameter EXPLANATION to ""
# R2.1.3 will not produce anything since CheckCondition3 will produce FALSE
# R2.1.4 will produce the atom C3_DONE
  <{CHECK_C3, C3_DONE}, (1, D3, "")>
# For branch 2 time 2:
# R2.2.1 will produce the atom EXPLAINED and set the parameter EXPLANATION to
# "condition 3 is not satisfied".
# R2.2.2 will not produce anything since C3_SAT is not in the previous state
# R2.2.3 will produce the atom END and will set DATA=null
  <{EXPLAINED, END}, (2, null, "condition 3 is not satisfied")>
```

For every branch the results can be obtained by examining states containing the atom END. If an explanation for a negative decision in a branch is found, then the atom EXPLAINED will be present in the state. In the case of our example we have two states with the atom END. Both of

these contain the atom EXPLAINED. Thus, the negative decision for all two branches is explained. The explanations are the values of the parameter EXPLANATION (3rd value in the generalized position tuple), which are "condition 2 is not satisfied" and "condition 3 is not satisfied".

5 Semantics of H-ASP PL

The semantics for H-ASP PL programs is a variant of the H-ASP stable model semantics. In order to diagnose failures of automatic whitelisting system, all the cases for whitelisting will need to be examined. Our H-ASP PL program will describe all of the whitelisting cases. We will be interested in a stable model of the underlying H-ASP program that will describe the results of examining each case. We would like this to be the unique maximal stable model. We will thus construct the semantics of H-ASP PL so that for a valid H-ASP PL program, its underlying H-ASP program has a unique maximal stable model, which after an appropriate transform, will be the stable model of the H-ASP PL program.

The semantics of H-ASP PL programs are defined in two steps. For a H-ASP PL program W, a transform $Tr\left[PL\right]$ is used to produce a H-ASP program $Tr\left[PL\right]\left(W\right)$. Then the transform Tr introduced in [4] is used to produce a H-ASP program $Tr\left(Tr\left[PL\right]\left(W\right)\right)$. $Tr\left(Tr\left[PL\right]\left(W\right)\right)$ has the following properties for an initial condition I of $Tr\left[PL\right]\left(W\right)$:
1. There is a bijection between the set of stable models of $Tr\left[PL\right]\left(W\right)$ with the initial condition I and the set of stable models of $Tr\left(Tr\left[PL\right]\left(W\right)\right)$ with the corresponding initial condition $J\left(I\right)$.
2. For the initial condition $J\left(I\right)$, $Tr\left(Tr\left[PL\right]\left(W\right)\right)$ has a unique maximal stable model $M'_{\max}$. $M'_{\max}$ is maximal in a sense that it contains all the stable models of $Tr\left(Tr\left[PL\right]\left(W\right)\right)$ with the initial condition $J\left(I\right)$.
Then we set the stable model of W with initial condition I to be the unique maximal stable model $M'_{\max}$ of $Tr\left(Tr\left[PL\right]\left(W\right)\right)$ with initial condition $J(I)$.

The transform $Tr\left[PL\right]$ is defined similarly to the transform $Tr\#$ introduced in [4] for transforming valid H-ASP# programs. The definitions of both transforms are omitted due to the space constraints. The following new theorem states that any computational cDAG representable by a computer can be computed by a H-ASP PL program.

Theorem 2. *Let At be a set of propositional atoms, and let S be a set whose elements are representable by a computer. Let $\langle V, E\rangle$ be a computational cDAG over At and S. Then there exists a H-ASP PL program P and an initial condition I for P such that $comp\left(Tr\left(Tr\left[PL\right]\left(P\right)\right),\ J\left(I\right)\right) = \langle W, U\rangle$ is isomorphic to $\langle V, E\rangle$.*

The theorem is proved by constructing an isomorphism based on a H-ASP PL program P and initial condition I, chosen so that $Tr\left(Tr\left[PL\right]\left(P\right)\right)$ and $J\left(I\right)$ are the H-ASP program and the corresponding initial condition constructed in the proof of theorem 1.

For a valid H-ASP PL program W and initial condition I of $Tr\left[PL\right]\left(W\right)$, the maximal stable model $M'_{\max}$ of $Tr\left(Tr\left[PL\right]\left(W\right)\right)$ with the initial condition $J\left(I\right)$ can be computed by the Local Algorithm [4].

An informal description of the Local Algorithm is as follows. The Local Algorithm is a multi-stage process where the hybrid states derived at stage n are used to derive hybrid states at stage $n+1$. Suppose that a hybrid state $(V, \mathbf{p})$ was derived by the Local Algorithm at stage n. The Local Algorithm first uses all the advancing rules applicable at $(V, \mathbf{p})$ to derive a set of the candidate next hybrid states $(Z_1, \mathbf{q}_1)$, ..., $(Z_k, \mathbf{q}_k)$. For each $(Z_i, \mathbf{q}_i)$ the stationary rules applicable at $(Z_i, \mathbf{q}_i)$

are then used to form an ASP program $D(Z_i, \mathbf{q}_i)$. Suppose that the stable models of $D(Z_i, \mathbf{q}_i)$ are Y_1, ..., Y_m (for each Y_j we have that $Z_i \subseteq Y_j$). Then the set of the next hybrid states with the generalized position $\mathbf{q}_i$ is $(Y_1, \mathbf{q}_i)$, ..., $(Y_m, \mathbf{q}_i)$.

Theorem 3. *(Based on theorem 82, [4]) For a valid H-ASP PL program W and initial condition I of $Tr[PL](W)$, the result of the Local Algorithm applied to W produces $M'_{\max}$, which is the unique maximal stable model of $Tr(Tr[PL](W))$ with the initial condition $J(I)$.*

6 Conclusion

The extensions of ASP that allow external data searches include DLV^{DB} system [25], VI programs [7], GRINGO grounder [12]. In [23], however Redl notes that HEX programs [11] can be viewed as a generalization of these formalisms. We will thus only describe the relation of our work to HEX programs.

HEX programs are an extension of ASP programs that allow accessing external data sources 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. The main similarities with this approach and our approach are that both H-ASP PL and HEX programs allow the use of external data sources, and both support mechanisms for passing the information between external algorithms. The main differences are the following: (1) in H-ASP PL the information processed by the external algorithms represents a type of information that is different from the information contained in the logical atoms, (2) the H-ASP PL programs have a built-in support for producing BCP cDAGs, and (3) a H-ASP program underlying the H-ASP PL definitions has a unique maximal stable model under H-ASP stable model semantics. The latter two properties make the H-ASP PL very convenient for the problem of diagnosing automatic whitelisting.

A relation of our approach for solving diagnostic problems to that of Balduccini and Gelfond (see [1]) was discussed in the beginning of this paper. We think that developing an approach similar to that of Balduccini and Gelfond for H-ASP is an interesting problem for future work.

In this paper we have discussed the use of H-ASP to diagnose failures of the automatic whitelisting system for Google's Dynamic Remarketing Ads. The software, which we discuss in this paper was used to diagnose and fix failures of the automatic whitelisting for many advertisers over a time interval of several months. Whereas the time needed to diagnose a single failure without the software was 30-60 minutes, the time needed to diagnose a single failure using the software was 1-3 minutes. The declarative nature of the H-ASP PL program made it easy to update the software so as to reflect multiple changes to automatic whitelisting system that occurred over time.

References

1. Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with a-prolog. *TPLP*, 3(4-5):425–461, 2003.
2. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
3. Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors. *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*. Springer, 2005.

4. Alex Brik. *Extensions of Answer Set Programming*. PhD thesis, UC San Diego, 2012.
5. Alex Brik and Jeffrey B. Remmel. Hybrid ASP. In John P. Gallagher and Michael Gelfond, editors, *ICLP (Technical Communications)*, volume 11 of *LIPIcs*, pages 40–50. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.
6. Alex Brik and Jeffrey B. Remmel. Computing a Finite Horizon Optimal Strategy Using Hybrid ASP. In *NMR*, 2012.
7. Francesco Calimeri, Susanna Cozza, and Giovambattista Ianni. External sources of knowledge and value invention in logic programming. *Ann. Math. Artif. Intell.*, 50(3-4):333–361, 2007.
8. Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The dlv system: Model generator and advanced frontends (system description). In *WLP*, pages 0–, 1997.
9. Deborah East, Mikhail Iakhiaev, Artur Mikitiuk, and Miroslaw Truszczynski. Tools for modeling and solving search problems. *AI Commun.*, 19(4):301–312, 2006.
10. Deborah East and Miroslaw Truszczynski. Predicate-calculus-based logics for modeling and solving search problems. *ACM Trans. Comput. Log.*, 7(1):38–83, 2006.
11. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 90–96. Professional Book Center, 2005.
12. Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.
13. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In Manuela M. Veloso, editor, *IJCAI*, pages 386–, 2007.
14. Michael Gelfond and Nicola Leone. Logic programming and knowledge representation - the a-prolog perspective. *Artif. Intell.*, 138(1-2):3–38, 2002.
15. Yuliya Lierler. cmodels - sat-based disjunctive answer set solver. In Baral et al. [3], pages 447–451.
16. Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.
17. Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. In Rina Dechter and Richard S. Sutton, editors, *AAAI/IAAI*, pages 112–118. AAAI Press / The MIT Press, 2002.
18. Lengning Liu and Miroslaw Truszczynski. Pbmodels - software to compute stable models by pseudo-boolean solvers. In Baral et al. [3], pages 410–415.
19. V. Wiktor Marek and Jeffrey B. Remmel. On the expressibility of stable logic programming. *TPLP*, 3(4-5):551–567, 2003.
20. V. Wiktor Marek and Jeffrey B. Remmel. Set constraints in logic programming. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pages 167–179. Springer, 2004.
21. V. Wiktor Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. pages 375–398, 1999.
22. Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.
23. Christoph Redl. *Answer Set Programming with External Sources: Algorithms and Efficient Evaluation*. PhD thesis, Vienna University of Technology, 2015.
24. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
25. Giorgio Terracina, Nicola Leone, Vincenzino Lio, and Claudio Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP*, 8(2):129–165, 2008.