A-Prolog as a tool for declarative programming

M. Balduccini[1], M. Gelfond[2], M. Nogueira[1]

[1] Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968
E-mail: {marcy,monica}@cs.utep.edu

[2] Department of Computer Science
Texas Tech University
Lubbock, TX 79409
E-mail: Michael.Gelfond@coe.ttu.edu

Abstract

In this paper we give a brief introduction to the declarative knowledge representation and logic programming language A-Prolog. We demonstrate the methodology of programming in A-Prolog by developing a simple declarative program describing dynamic behavior of combinational digital circuits. The implementation is proven to be correct and is supplied with a graphical interface which facilitates the use by students. Our experiment confirms our belief that A-Prolog can become a language of choice for various knowledge intensive applications.

1. Introduction

It is becoming increasingly clear that to fully realize the potential of the computer revolution, computer scientists must develop a systematic methodology for design and construction of software systems capable of basing their behavior on knowledge about their environment. Without such methodology we can create neither autonomous robots nor intelligent information, expert, and decision making systems. This realization led to work on design and implementation of powerful knowledge representation languages ([3, 17]).

Most of these languages are based on various forms of classical logic. Even though useful in many situations, these languages are severely limited in their ability to represent various forms of common sense knowledge. In particular, they are not very suitable to represent and reason about defaults, i.e., statements of the form "Elements of class C *normally, as a rule* have property P." Defining the sets of valid conclusions which can be obtained from collections of defaults, and discovering various ways to efficiently arrive at such conclusions, is long recognized as one of the central themes in the area of knowledge representation and reasoning. Recent advances on solution of this problem led to development of knowledge representation languages with roots in logic programming. Unlike classical logic, the logic which forms the basis for these languages is *n*onmonotonic, i.e., allows the reasoner to withdraw his/her previous conclusions when new information becomes available. This property of the logic allows an elegant formalization of default reasoning. In this paper we give a brief introduction to one of such languages, called A-Prolog. We believe that A-Prolog is rapidly becoming a very promising candidate as the language of choice for many knowledge representation tasks [19, 26] (for an alternative approach see [1]). The use of A-Prolog will be illustrated by developing a simple declarative program describing dynamic behavior of combinational digital circuits.[1]

The program is short and has mainly an illustrative character. Its natural extensions can, however, be used as a learning tool by students taking classes in digital logic.

The syntax and semantics of A-Prolog is not new. The syntax expands a traditional notion of a rule from "classical" logic programming ([16]) by introducing additional logical connectives. The semantics is given by the notion of the answer set of a program [10, 11]. Originally, answer set semantics of "classical" logic programs were designed to give a declarative meaning to the logical connective *not*, called negation as failure, used in the Prolog programming language. Later, the language was extended by adding classical negation, disjunction, and other constructs, which made it a powerful theoretical tool for studying various forms of common sense reasoning (see, for instance, [4]). Practical applications of A-Prolog were somewhat limited by the lack of a powerful inference engine capable of computing its entailment relation. The "classical" logic programming engines like those implemented in Prolog and XSB interpreters ([32]), though sound with respect to answer set semantics, are not sufficiently powerful to

[1] In this paper, we refer to "combinational digital circuits" as "digital circuits."

reason with programs which have more than one answer set. This situation arises when knowledge about the domain, encoded by the corresponding program, is incomplete and allows alternative consistent views of what relations are satisfied by the objects of the domain. Such incompleteness causes Prolog-like engines to go into infinite loops or to return the answer *unknown* to too many queries. The situation changed recently with the coming of age of a new class of inference engines for A-Prolog, such as SMODELS ([25]), DLV ([8]), DeReS ([5]). These engines are limited to programs which have a finite number of answer sets. Given a program they compute one or all of such models. In this sense, the engines can be better viewed as satisfiability checkers than as theorem provers. Their development substantially increased the use of A-Prolog for various applications [6, 7, 9, 14, 37] and allowed some authors to talk about a new logic programming paradigm [19, 26].

The paper is organized as follows. In the next section we introduce syntax and semantics of A-Prolog. Section 3 gives a short introduction to digital circuits and section 4 presents the simple circuit theory in A-Prolog. In section 5 we show how this theory can be applied to solve several problems in circuit design. Conclusions and future work are presented in the last section.

2. A-Prolog

In this paper the language of choice is A-Prolog – a language of logic programs under the answer set semantics [10, 11]. A program in A-Prolog consists of a signature Σ and a collection of rules of the form:

$$head \leftarrow body \tag{1}$$

where *head* is empty or consists of a literal l_0 and *body* is of the form: $l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$ where l_i 's are literals over Σ. A literal is an atom p or its negation $\neg p$. If *body* is empty we replace $\leftarrow$ by a period. While $\neg p$ says that p is false, *not* p has an epistemic character and can be read as "there is no reason to believe that p is true." The symbol *not* denotes a non-standard logical connective often called *default negation* or *negation as failure*. A program Π in A-Prolog can be viewed as a specification given to a rational agent for constructing beliefs about possible states of the world. Technically, these beliefs are captured by the notion of *answer set* of program Π. By $ground(\Pi)$ we denote a program obtained from Π by replacing variables by the ground terms of Σ. By answer sets of Π we mean answer sets of $ground(\Pi)$. If Π consists of rules not containing default negation, then its answer set S is the smallest set of ground literals of Σ which satisfies the following two conditions:

1. S is closed under the rules of $ground(\Pi)$, i.e., for every rule (1) in Π, either there is a literal l in its body such that $l \notin S$ or its non-empty head $l_0 \in S$.

2. If S contains an atom p and its negation $\neg p$, then S contains all ground literals of the language.

It is not difficult to show that there is at most one set $(Cn(\Pi))$ satisfying these conditions.

Now let Π be an arbitrary ground program in A-Prolog. For any set S of ground literals of its signature Σ, let Π^S be the program obtained from Π by deleting:

(i) each rule that has an occurrence of *not* l in its body with $l \in S$,

(ii) all occurrences of *not* l in the bodies of the remaining rules.

Then S *is an answer set of* Π *if*

$$S = Cn(\Pi^S). \tag{2}$$

In this paper we limit our attention to *consistent* programs, i.e., programs with at least one consistent answer set. Let S be an answer set of Π. A ground literal l is *true* in S if $l \in S$; *false* in S if $\neg l \in S$. This is expanded to conjunctions and disjunctions of literals in a standard way. A query Q is *entailed* by a program Π ($\Pi \models Q$) if Q is true in all answer sets of Π. Queries $l_1 \wedge \ldots \wedge l_n$ and $\overline{l_1} \vee \ldots \vee \overline{l_n}$ are called complementary.[2] If Q and $\overline{Q}$ are complementary queries, then Π's answer to Q is *yes* if $\Pi \models Q$; *no* if $\Pi \models \overline{Q}$, and *unknown* otherwise.

Here are some examples. Assume that signature Σ contains two object constants a and b. The program $\Pi_1 = \{\neg p(X) \leftarrow not\ q(X).\ \ q(a).\}$ has the unique answer set $S = \{q(a), \neg p(b)\}$. The program $\Pi_2 = \{p(a) \leftarrow not\ p(b).\ \ p(b) \leftarrow not\ p(a).\}$ has two answer sets, $\{p(a)\}$ and $\{p(b)\}$. The programs $\Pi_3 = \{p(a) \leftarrow not\ p(a).\}$ and $\Pi_4 = \{p(a).\ \ \leftarrow p(a).\}$ have no answer sets.

It is easy to see that programs of A-Prolog are nonmonotonic. For example consider program Π_1. We saw that $\Pi_1 \models \neg p(b)$, however, if some new information, $q(b)$, is added to the program, it forces the withdrawal of the previous conclusion $\neg p(b)$. The new program $\Pi_1 \cup q(b)$ has the following unique answer set $\{q(a), q(b)\}$. Nonmonotonicity is an important feature

[2] Given an atom p, $\bar{l}$ is defined as $\neg p$ if $l = p$, or as p if $l = \neg p$.

of A-Prolog which makes it a suitable formalism for representing common sense reasoning and reasoning about time and change. A-Prolog is closely connected with more general nonmonotonic theories. In particular, as was shown in [11, 18], there is a simple and natural mapping of programs in A-Prolog into a subclass of Reiter's default theories [29]. Similar results are also available for Autoepistemic Logic [23].

3. Digital Circuits

Normally, computer science students start to study foundations of digital design in their first or second year at the university. First they concentrate on combinational circuits which are constructed from simple boolean gates and are used to compute boolean functions. Given such a function $Y = f(X_1, \ldots, X_n)$, where Y and $X_1, \ldots, X_n$ are boolean variables, students learn how to use propositional logic to construct a circuit C which *instantaneously* transforms the values $X_1, \ldots, X_n$ applied on its input wires $W_1, \ldots, W_n$ to the value Y on its output wire W. Later, they move to building more complex devices employing more complex, sequential circuits. The model of a circuit remains, however, essentially boolean with the only possible signals corresponding to 0 and 1, and basic gates still performing instantaneous transmission of information. In more advanced classes students normally "discover" that the boolean model they have learned is not always a realistic one. Gates suffer from physical limitations, i.e., do not instantaneously perform the function that they implement because of propagation (and other types) of delays. For a short time, the values of signals may lie somewhere between the levels necessary to classify them as 0 and 1, and will therefore be undefined. There are other situations where the analog (continuous and non-digital) character of gates and signals should be taken into account. To model such phenomena, scientists introduced the notion of a digital circuit with delays ([22, 36]) and three possible input values: 0, 1, and 1/2 (undefined) [36]. These circuits do not instantaneously produce the values of the corresponding functions. Instead, these values are produced after delays, which are determined by the circuit and the vector of input signals.

To make it usable for mathematical proofs, this explanation needs to be clarified.

Definition 3.1 *Let $X_1, \ldots, X_n$ be boolean values applied to input wires $W_1, \ldots, W_n$ of a circuit C at time t, let $Y = f(X_1, \ldots, X_n)$ be a boolean function, and δ be a non-negative integer. We say that circuit C computes $f(X_1, \ldots, X_n)$ with a delay δ if, in the absence of other inputs, the value on its output wire W at any time $t' \geq t + \delta$ is equal to Y.* C computes *function f with a delay δ if it computes all the values of f with this delay.*

Introduction of delays and unknown signals bring to life a number of questions not present in the case of *ideal* (time independent) boolean circuits. We need to know for instance, how these δ's can be computed, how we can guarantee that a particular circuit computes f with a given δ, how we can check if a component of a circuit can be replaced by a similar component with a smaller/bigger delay without violating some important properties of the circuit, etc. To answer these and similar questions we need to have a precise description of the behavior of a circuit, which, given a sequence of values applied to its input wires, will determine the values of signals present on *every* wire of the circuit at any moment of time. In the next section we will design and implement a program in A-Prolog which does exactly that.

4. Circuits in A-Prolog

We will start with introducing a simple language L for describing digital circuits. The language has four types of object constants (names for objects of the domain): (a) $g_1, g_2 \ldots$ for gates; (b) $w_1, w_2, \ldots$ for wires; (c) $0, 1, u$ for signals; (d) $andg$, org, $notg$ for the three basic gate types we chose to represent.

Variables for gates, wires, and signals will be denoted by possibly indexed letters G, W and S respectively. We will also assume that L contains standard notation for numbers, needed to denote delays. To describe the geometry of the circuit we use statements of the form $output(W, G)$ and $input(W, G)$ read as "W is an output (input) wire of gate G." The types of gates in the circuit and the gates' delays are expressed by the statements $type\_of(G, gate\_type)$ (G is of type $gate\_type$) and $delay(G, D)$ (G has delay D). In this notation, the circuit from fig.1 corresponds to the following collection of statements of A-Prolog:

$input(w_1, g_1)$.	$type\_of(g_1, notg)$.
$input(w_2, g_2)$.	$type\_of(g_2, notg)$.
$input(w_3, g_3)$.	$type\_of(g_3, andg)$.
$input(w_4, g_3)$.	$delay(g_1, 1)$.
$output(w_3, g_1)$.	$delay(g_2, 0)$.
$output(w_4, g_2)$.	$delay(g_3, 1)$.
$output(w_5, g_3)$.	

We denote such a representation of a circuit C by $\pi(C)$. To simplify the user/program interface we implemented a schematic entry program which allows the user to draw a circuit diagram by choosing from the

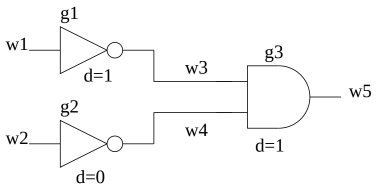

Figure 1. Graphical representation of a digital circuit.

options available on the ToolBox Window (shown in fig.2(a)). For example, fig.5(b) shows how the circuit diagram presented in fig.1 will appear on the graphical interface. Once the circuit drawing is complete, the entry program automatically translates it into the corresponding A-Prolog representation.

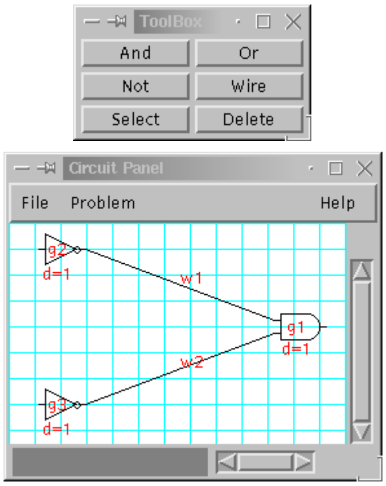

Figure 2. (a) ToolBox Window. (b) The complete circuit.

To describe the dynamic behavior of the circuit we need to introduce the notion of time. AI researchers developed a large variety of different models of time. For our purposes, we assume the discrete linear time model in which time is represented by non-negative integers. We will view the application of signals to the input wires of a circuit as the execution of an action which changes the previous signals on these wires. This triggers a process of signal propagation through the circuit which goes uninterrupted unless the input signals are changed again. In this way, describing behavior of the circuit can be reduced to specifying effects of the corresponding actions as it is done in action theories of AI (see for instance [12, 15, 20, 27, 30, 35]). In these theories, dynamic domains consist of actions and fluents (functions whose values depend on time). Action theories are built to specify the values of fluents at an arbitrary moment t, given their values at moment 0 and the domain history (a sequence of actions performed in the domain in the past). In our domain we have only one (parameterized) action $apply(w, s)$ and one (parameterized) propositional fluent, $value(w, s)$. A statement $occurs(apply(w, s), t)$ says that at moment t signal s is applied to wire w, while a statement $h(value(w, s), t)$ denotes that at moment t the value of the signal on w is s. We will also need an auxiliary relation $opposite(s_1, s_2)$ satisfied by the pairs $[0, 1]$, $[1, 0]$ and $[u, u]$. Effects of actions will be represented in A-Prolog by the following rule:

$$h(value(W, S), T) \quad \leftarrow \quad occurs(apply(W, S), T).$$

Here T is a variable for time. To guarantee the computability of our models we assume that T ranges between 0 and some fixed time denoted by the constant $last\_time$. (This constant can be viewed as a parameter of our system and it is entered by the user via the entry program as a part of the problem instance.) The next rule describes the propagation of the applied signal through the *not* gate of the circuit.

$$\begin{array}{lcl} h(value(W_2, S_1), T + D) & \leftarrow & type\_of(G, notg), \\ & & delay(G, D), \\ & & input(W_1, G), \\ & & output(W_2, G), \\ & & opposite(S_1, S_2), \\ & & h(value(W_1, S_2), T). \end{array}$$

To represent the function of gates AND and OR, we need to define some auxiliary relations. The first relation, $not\_all(G, S, T)$, holds if at moment T some input wire of the gate G has a signal different from S. This can be expressed by the following rule:

$$\begin{array}{lcl} not\_all(G, S_1, T) & \leftarrow & input(W, G), \\ & & S_1 \neq S_2, \\ & & h(value(W, S_2), T). \end{array}$$

The second relation, $all(G, S, T)$, holds if at time T all the input wires of G have value S, and is defined by the rule:

$$all(G, S, T) \quad \leftarrow \quad not\ not\_all(G, S, T).$$

Finally, the relation $contains(G, S, T)$ holds if at moment T at least one input wire of G has value S, and is defined by the rule:

$$\begin{array}{lcl} contains(G, S, T) & \leftarrow & input(W, G), \\ & & h(value(W, S), T). \end{array}$$

Now we can define the propagation of signals through *and* gates:

$$
\begin{array}{lcl}
h(value(W,1),T+D) & \leftarrow & type\_of(G,andg), \\
& & delay(G,D), \\
& & output(W,G), \\
& & all(G,1,T). \\
h(value(W,0),T+D) & \leftarrow & type\_of(G,andg), \\
& & delay(G,D), \\
& & output(W,G), \\
& & contains(G,0,T). \\
h(value(W,u),T+D) & \leftarrow & type\_of(G,andg), \\
& & delay(G,D), \\
& & output(W,G), \\
& & not\ contains(G,0,T), \\
& & contains(G,u,T).
\end{array}
$$

The corresponding rules for the *or* gates can be defined similarly. All the above rules define the effects of changes caused in the circuit by applying new signals to its input wires. To complete our program we need to specify when the values of fluents do not change. The task of finding a compact way to specify this in a formal language is called the *frame problem*.[3] J. McCarthy in [21] suggested that this problem is closely related to the problem of representing a particular default called the *law of inertia*. The law says that "normally, things stay as they are", i.e., in dynamic domains fluents do not change their values unless they are forced to. Fortunately, the methodology of representing defaults in A-Prolog is now well understood and can be applied to obtain a simple and natural solution of the frame problem for our domain. The solution is given by the next two rules.

The first of them is the Law of Inertia:

$$
\begin{array}{lcl}
h(value(W,S),T+1) & \leftarrow & h(value(W,S),T), \\
& & not\ \neg h(value(W,S),T+1).
\end{array}
$$

This rule allows the reasoner (the program) to assume that the value of a signal on a wire W does not change from one moment to the next, unless it is forced to believe otherwise. The second rule states that there may be at most one signal present on a wire at a given moment of time:

$$
\begin{array}{lcl}
\neg h(value(W,S_1),T) & \leftarrow & S_1 \neq S_2, \\
& & h(value(W,S_2),T).
\end{array}
$$

We denote the resulting program by CT and call it the *simple circuit theory*. The theory, in conjunction with the specification of a circuit and its history up to the current moment t_c, can be used to specify the values of signals on the circuit wires at an arbitrary moment $0 \leq t \leq last\_time$. We call such a specification a *domain description* at time t_c. It consists of the encoding of a circuit in language L (see fig.1) together with statements of the form:

$$occurs(apply(w,s),t).$$

where $0 \leq t \leq t_c$. Our schematic entry program allows the user to specify the input graphically. We assume that, unless otherwise specified, the initial signals of the circuit are unknown. This assumption can be represented in A-Prolog by the rules:

$$
\begin{array}{lcl}
h(value(W,u),0) & \leftarrow & not\ known\_value(W,0). \\
known\_value(W,0) & \leftarrow & h(value(W,1),0). \\
known\_value(W,0) & \leftarrow & h(value(W,0),0).
\end{array}
$$

To simplify our presentation we just add these rules to our CT theory.

We assume that domain descriptions used in conjunction with CT are consistent, i.e., do not contain physical impossibilities such as: two different signals applied to the same wire at the same time, multiple input wires for the *not* gate, etc. Our schematic entry program will eliminate the possibility of inconsistent data to be entered into the corresponding domain description.

Using standard mathematical techniques recently developed by researchers in logic programming and nonmonotonic reasoning, it is not difficult to show that for any consistent domain description $\mathcal{D}$, the program $CT \cup \mathcal{D}$ has exactly one consistent answer set. By $CT(\mathcal{D})$ we denote the set of all atoms, formed by predicate symbol h, which belong to this answer set. The set $CT(\mathcal{D})$ can be viewed as a *specification of a dynamic behavior of a combinational circuit with delays.*[4] Let us first show that our specification correctly captures the behavior of "ideal" combinational circuits.

Proposition 4.1 *Let C be a combinational circuit, with input wires $w_1, \ldots, w_n$, output wire w, and no delays, which computes a boolean function $f(S_1, \ldots, S_n)$. Then for any input vector $s_1, \ldots, s_n$ of 0's and 1's, $h(value(w,s),0) \in CT(\mathcal{D})$ iff $s = f(s_1, \ldots, s_n)$, where $\mathcal{D} = \pi(C) \cup \{occurs(apply(w_i,s_i),0) : 1 \leq i \leq n\}$.*

Any combinational circuit C with delays has its *ideal counterpart*, $i(C)$ obtained from C by setting all of the gate delays of C to 0. The following proposition guarantees that for any input vector, $s_1, \ldots, s_n$, the output signal of C will eventually stabilize at the value of $f(s_1, \ldots, s_n)$ where f is the function defined by the ideal counterpart of C. More precisely,

[3] The problem, posed in 1969 by J. McCarthy, proved to be a difficult one and stimulated several interesting lines of research in AI ([33]), particularly in research on nonmonotonic logics, logic programming and theories about action and change.

[4] The above definition works only for circuits computing a "single value" boolean function, i.e., a function returning 0 or 1. This restriction is only for simplicity of presentation. All the definitions and programs can be easily extended to functions returning vectors of boolean values.

Proposition 4.2 *Let C be a combinational circuit with input wires $w_1, \ldots, w_n$ and output wire w, and let $f(S_1, \ldots, S_n)$ be a function computed by its ideal counterpart $i(C)$. Then there is a delay, δ, such that for any $t \geq \delta$ and any input vector $s_1, \ldots, s_n$ of 0's and 1's, $h(value(w,s),t) \in CT(\mathcal{D})$ iff $s = f(s_1, \ldots, s_n)$, where $\mathcal{D} = \pi(C) \cup \{occurs(apply(w_i, s_i), 0) : 1 \leq i \leq n\}$.*

The circuit delay from the above proposition can be found constructively. This can be done by another A-Prolog program, Δ, consisting of the following rules:

$$
\begin{array}{lll}
is\_input\_wire(W) & \leftarrow & input(W,G), \\
 & & not\ is\_output(W). \\
is\_output(W) & \leftarrow & output(W,G). \\
is\_output\_wire(W) & \leftarrow & output(W,G), \\
 & & not\ is\_input(W). \\
is\_input(W) & \leftarrow & input(W,G). \\
in\_gate(G) & \leftarrow & is\_input\_wire(W), \\
 & & input(W,G). \\
out\_gate(G) & \leftarrow & is\_output\_wire(W), \\
 & & output(W,G). \\
out\_delay(G,N) & \leftarrow & in\_gate(G), \\
 & & delay(G,N). \\
in\_delay(G_2,N) & \leftarrow & output(W,G_1), \\
 & & input(W,G_2), \\
 & & out\_delay(G_1,N). \\
\neg max\_in\_delay(G,N) & \leftarrow & in\_delay(G,N), \\
 & & in\_delay(G,M), \\
 & & M > N. \\
max\_in\_delay(G,N) & \leftarrow & in\_delay(G,N), \\
 & & not\ \neg max\_in\_delay(G,N). \\
out\_delay(G,N) & \leftarrow & max\_in\_delay(G,N_1), \\
 & & delay(G,N_2), \\
 & & N = N_1 + N_2. \\
circuit\_delay(N) & \leftarrow & out\_gate(G), \\
 & & out\_delay(G,N).
\end{array}
$$

The program defines a simple algorithm for computing circuit delays which can be found in standard introductory texts on digital logic ([13, 31, 36]). The result is not necessarily optimal, but it may serve as a good practical approximation. (It is instructive to notice how rules of A-Prolog are used to encode recursive definitions.) Again, it is not difficult to show that the program $\Delta \cup \pi(C)$ has exactly one answer set and that the answer set contains exactly one atom formed by the predicate symbol *circuit_delay*. Let us denote this atom by $circuit\_delay(d)$. We call number d the *computed delay* of C and denote it by $\delta(C)$.

Now we can prove the following proposition.

Proposition 4.3 *Let C be a combinational circuit with input wires $w_1, \ldots, w_n$ and output wire w and let $f(S_1, \ldots, S_n)$ be a function computed by its ideal counterpart $i(C)$. Then for any $t \geq \delta(C)$ and any input vector $s_1, \ldots, s_n$ of 0's and 1's, $h(value(w,s),t) \in CT(\mathcal{D})$ iff $s = f(s_1, \ldots, s_n)$, where $\mathcal{D} = \pi(C) \cup \{occurs(apply(w_i, s_i), 0) : 1 \leq i \leq n\}$.*

5. Using CT

The discussion in the previous section was limited to the use of declarative semantics of A-Prolog for specifying the behavior of digital circuits. Thanks to the existence of inference engines for A-Prolog, this specification can be combined with simple reasoning programs aimed at solving various design tasks, and it can also be actually executed. We believe that the resulting executable prototypes can help to clarify and better understand some notions related to digital circuits. It remains to be verified if its applicability can go beyond classrooms and theoretical laboratories. In this section we will give some examples of such prototypes.

5.1. Simulating the circuit

In many cases it may be instructive for a student to see the simulated behavior of the circuit. Ideally, this should be an easy task: the student will specify the circuit and its history using our graphical interface. The corresponding domain description $\mathcal{D}$, combined with CT, will be given as an input to one of the A-Prolog inference engines, say SMODELS, which will compute the program's unique answer set. The circuit behavior defined by $CT(\mathcal{D})$ will be extracted from the answer set and displayed in graphical and numerical form on the screen. The reality is rather close to the ideal situation, but not identical to it. The reason is that different inference engines have different restrictions on the programs needed to guarantee their soundness and completeness with respect to the semantics of A-Prolog. This implies that CT (from the previous section) needs to be slightly modified for the use of SMODELS. Fortunately, the modification is simple and basically amounts to replacing our typed variables by the explicit types (see [2] for details.)

After this modification is done, the resulting system will produce the output shown in fig.3, when given the description of the circuit from fig.1 and the following sequence of input values: $[0,0]$ applied on $[w_1, w_2]$ at time 0, and 1 applied on w_1 at time 1.

The output shows the propagation of symbols through the circuit up to moment 10. This graphical represen-

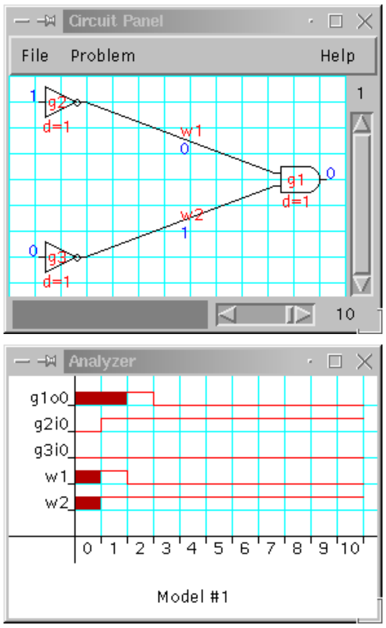

Figure 3. (a) Output in numerical form. (b) Timing Analysis.

tation helps the student to visualize and better understand the dynamic behavior of the circuit.

5.2. Avoiding hazards

One interesting problem when dealing with digital circuits involving delays is the occurrence of transient incorrect signal values, called *glitches*, on some of the circuit wires. A hazard is said to exist when a circuit has a possibility of producing such a glitch. A logic designer must be prepared to eliminate hazards even though a glitch may occur only under the worst-case combination of logical and electrical conditions [36]. We briefly describe a declarative program for the detection of a particular form of hazard. Combined with the inference engine of SMODELS this gives us a new algorithm for finding hazards different from the known algorithms (see [24]). Again, we believe that the program is sufficiently clear and the algorithm is reasonably efficient to help a student to understand the phenomena.

We will say that a circuit C, computing boolean function f is *hazardous* if there are two vectors, I_1 and I_2, of input signals which differ on the value of exactly one input wire[5], and during the transition period the value on the output wire of C changes to a signal different from $f(I_2)$.

[5] We call a consecutive application of such input signals to C a *simple transition*.

To better understand this notion let us consider the circuit in fig.4, taken from [36].

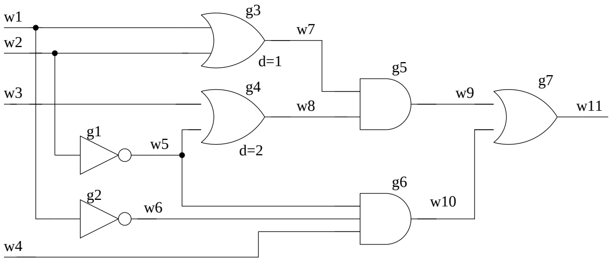

Figure 4. Circuit with a hazard.

In this circuit, there are 3 paths from input wire w_2 to output wire w_{11}. We assume that all gates, except g_3 and g_4, have delay 0. The delay of g_3 is 1 and the delay of g_4 is 2. Two of the paths go through these slower gates and affect the output signal. To understand how, let us consider the following evolution of the circuit signals. (a) Applying input signals $[0,0,0,1]$ to input wires $[w_1,w_2,w_3,w_4]$ causes the output signal to become 1 at time 0. If we change (b) the value on input wire w_2 to 1 at time 1, this change is propagated through the circuit and makes the output value of C become 0 at time 1. However, the output value of gate g_3 is delayed by 1 time unit and (c) will force the output of the circuit to change again to 1 at time 2. Then, (d) the output of the slower gate g_4, with delay 2, also changes, forcing the circuit output signal to finally reach value 0 at time 3. Therefore, a single transition on input w_2 caused the values of output w_{11} to change three times, as follows: $1 \to 0 \to 1 \to 0$.

Our goal now is to define hazardous circuits in *A-Prolog*. We will construct a program, GD (which stands for "glitch detector"), such that $GD \cup \pi(C)$ will have an answer set iff a circuit C is hazardous.

We assume that w is the output wire of circuit C and that there is a relation $required\_output(s)$ such that for any domain description $\mathcal{D}$, $required\_output(s)$ belongs to the answer set of $CT \cup \mathcal{D}$ iff s is the output signal of the ideal counterpart $i(C)$ of C. (To simplify the presentation we skip the definition of this relation which can be found in [2].) Suppose now we are given a history H containing a simple transition from I_1 to I_2. Then, *by definition*, this transition causes a glitch if the following condition holds:

$$
\begin{array}{lll}
glitch & \leftarrow & required\_output(S_1), \\
 & & h(value(w,S_1),T_1), \\
 & & h(value(w,S_2),T_2), \\
 & & S_1 \neq S_2, \\
 & & T_2 > T_1.
\end{array}
$$

Adding the above rule together with a constraint

$$\leftarrow \quad not\ glitch.$$

to GD will ensure that if C is safe (i.e., has no hazard) then $GD \cup \pi(C)$ will have no answer set.

To complete the construction of GD we need to generate histories containing possible simple transitions and check that they do not contain glitches. This can be done by first generating possible input vectors applied to C at moment 0, which is achieved by the rules:

$$\begin{aligned} occurs(apply(W,1),0) \leftarrow\ & is\_input\_wire(W), \\ & not\ occurs(apply(W,0),0). \end{aligned}$$

$$\begin{aligned} occurs(apply(W,0),0) \leftarrow\ & is\_input\_wire(W), \\ & not\ occurs(apply(W,1),0). \end{aligned}$$

Then, we proceed by introducing a new relation $change(W)$ which holds when at moment 1 the signal applied to wire W at 0 is changed to its opposite.

$$\begin{aligned} occurs(apply(W,S_1),1) \leftarrow\ & change(W), \\ & occurs(apply(W,S_2),0), \\ & opposite(S_1,S_2). \end{aligned}$$

To ensure that histories generated by our program contain only simple transitions we need to add the following rule:

$$change(w_1) \text{ or } \ldots \text{ or } change(w_k),$$

where $w_1, \ldots, w_k$ is the list of the input wires of C. The DLV engine would understand this rule and would properly compute the corresponding answer sets. However, to make it work for SMODELS we need to eliminate the disjunction, which can be done by the following rules:

$$\begin{aligned} change(W) \quad & \leftarrow \quad is\_input\_wire(W), \\ & \qquad\ not\ other\_changed(W). \\ other\_changed(W) \quad & \leftarrow \quad change(W_1) \\ & \qquad\ W \neq W_1. \end{aligned}$$

Let GD be the program consisting of the rules of CT, which were introduced in this subsection, and the definition of the relation *required_output*.

Proposition 5.1 *A combinational circuit C is hazardous iff $GD \cup \pi(C)$ is consistent, i.e., has an answer set.*

Notice that each answer set describes a simple transition causing a glitch and the signals propagation through the circuit. Our graphical interface allows the user to specify a circuit and request it to be checked for glitches. For example, in the circuit C from fig.4, the program will return a message box informing the user that the circuit is hazardous, and it will also graphically show, via the Analyzer Window, the situations in which the glitch occurs, (see fig.5.)

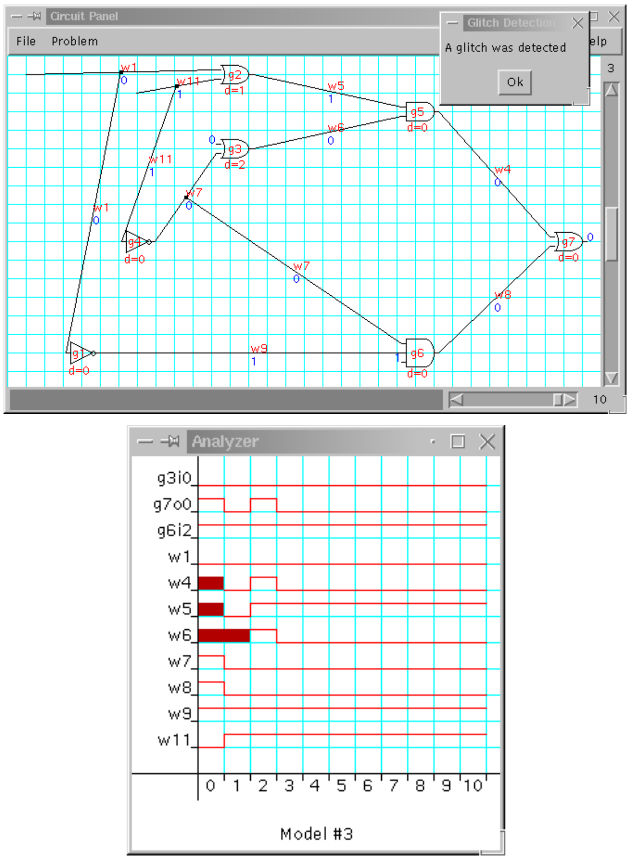

Figure 5. Interface output for glitch detection problem.

The simple theory for circuits, CT, can be used in a similar way to solve other problems associated with digital design. CT, along with various reasoning modules, can be used to decide what signals should be applied to the input wires of a circuit to produce the desired output, to find malfunctioning components responsible for the incorrect behavior of a circuit, to simulate certain forms of sequential circuits, etc. We hope, however, that the above examples are sufficient for illustrative purposes.

6. Conclusions

It was believed for some time that A-Prolog is capable of representing default knowledge as well as various forms of knowledge incompleteness. Quite recently it was noticed that A-Prolog is also suitable for modeling reasoning of agents in dynamic domains. Even more recently, it was understood that methodologies of declarative programming developed in these two areas can be used in many other interesting domains. In this paper we gave a short introduction to the syntax and semantic of A-Prolog and demonstrated the applicability of this methodology by solving the problem of reasoning about digital circuits. The resulting system was used by some of our students and we are planning to make it available for general use in some of the related classes.

When modeling complex domains, the syntactic restrictions of A-Prolog can make some rules appear non-natural – in our case, the three rules used in GD in order to exhaustively generate possible input vectors for the circuit. We are currently working on an extension of A-Prolog to deal with sets that is expected to overcome this problem.

We would like to stress the following software engineering lessons learned from this experiment:

1. syntax and semantic of A-Prolog, as well as its mathematical theory, allowed us to quickly build a concise and modular solution to a comparatively non-trivial problem;

2. the solution was constructed in parallel with the development of the proof of its correctness. Declarativeness of A-Prolog greatly facilitated this process;

3. reasoning and constraint satisfaction algorithms built in $theA$-Prolog inference engine proved to be sufficiently efficient for implementing interesting new algorithms for simulation and analysis of digital circuits. Comparison of their efficiency with respect to other known algorithms remains to be investigated;

4. declarative programs in A-Prolog were nicely integrated with each other and with the Java-based graphical interface allowing a user–friendly interaction with the system.

We believe that the integration of programs written in different languages, with different programming paradigms, will be a trademark of future knowledge intensive systems.

7. Acknowledgments

The authors wish to thank Ramon P. Otero and Alessandro Provetti for useful discussions on the subject of this paper, Mauro Ferrari and Vladimir Lifschitz for pointing us to related information sources, and Nelly Delgado for proof–reading an early draft of the paper.

Marcello Balduccini was partially supported by "Fondazione f.lli Confalonieri" and by a NASA contract. Monica Nogueira was partially supported by United Space Alliance Reasoning Research Grant under contract 2635-0221.

References

[1] J. J. Alferes and L. M. Pereira *Reasoning with Logic Programming.* LNAI volume 1111, Springer-Verlag, 1996.

[2] M. Balduccini, M. Gelfond and M. Nogueira. Reasoning about Digital Circuits in A-Prolog. *Technical Report.* The University of Texas at El Paso and Texas Tech University, 2000.

[3] P. Clark and B. Porter. KM – the knowledge machine: Reference manual. Technical report, AI Lab, University of Texas at Austin, 1998.

[4] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 12:1-80, 1994.

[5] P. Cholewinski, W. Marek and M. Truszczyński. Default Reasoning System DeReS. In *Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pp. 518-528. Morgan Kauffman, 1996.

[6] Y. Dimopoulos, B. Nebel, and J. Koehler Encoding Planning Problems in Nonmonotonic Logic Programs. In *European Workshop on Planning*, 1997.

[7] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Databases. *ACM Transactions on Database Systems*, 22(3), pp. 364-418, Sept. 1997.

[8] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System dlv: Progress Report, Comparisons and Benchmarks. In A. G. Cohn, L. Schubert, and S. C. Shapiro, (eds), *Proc. of KR'98*, pp. 406-417. Morgan Kaufmann Publishers, 1998.

[9] E. Erdem, V. Lifschitz, and M. Wong. Wire routing and satisfiability planning. Unpublished draft, 2000. (Available at *http://www.cs.utexas.edu/users/esra/papers.html*)

[10] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *Proc. of the 5th Int'l Conf. on Logic Programming*, pp. 1070-1080, 1988.

[11] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365-386, 1991.

[12] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, Vol. 3, No. 16, 1998.

[13] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Computer Science Series, 1978.

[14] V. Lifschitz. Action languages, Answer Sets, and Planning. In *The Logic Programming Paradigm: a 25-Year Perspective*. Spring-Verlag, 1999.

[15] F. Lin. Embracing causality in specifying the indirect effects of actions. *Proc. of IJCAI 95*, pp. 1985-1991, 1995.

[16] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag Symbolic Computation Series, 1987.

[17] R. MacGregor and R. Bates. The LOOM knowledge representation language. Technical Report ISI-RS-87-188, ISI, CA, 1987.

[18] W. Marek and M. Truszczyński. Stable semantics for logic programs and default theories. In *Proc. of the North American Conference on Logic Programming*, pp. 243-256. MIT Press, 1989.

[19] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pp. 375-398. Springer-Verlag, 1999.

[20] N. McCain and H. Turner. A causal theory of ramifications and qualifications. *Proc. of IJCAI 95*, pp. 1978-1984, 1995.

[21] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence*, Vol. 4, pp. 463-502. Edinburgh University Press, 1969.

[22] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Series in Electrical and Computer Engineering, 1994.

[23] R. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75-94, 1985.

[24] E. J. McCluskey. Introduction to the Theory of Switching Circuits. McGraw-Hill, 1965.

[25] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proc. of Joint Int'l Conf. and Symposium on Logic Programming*, pp. 289-303. MIT Press, 1996.

[26] I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In *Proc. of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pp. 72-79, 1998.

[27] J. Pinto and R. Reiter. Reasoning about Time in the Situation Calculus. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):251-268, September, 1995.

[28] T. Przymusinski. The Well-Founded Semantics Coincides With The Three-Valued Stable Semantics, *Fundamenta Informaticae*, 13:445-464, 1990.

[29] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81-132, 1980.

[30] R. Reiter Natural actions, concurrency and continuous time in the situation calculus. In L.C. Aiello, J. Doyle, and S.C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proc. of KR'96*, pp. 2-13. Morgan Kaufmann Publishers, 1996.

[31] C. H. Roth, Jr. *Fundamentals of Logic Design.* West Publishing Company, 1992.

[32] K. Sagonas, T. Swift, D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pp. 442-453, 1994.

[33] M. Shanahan. *Solving the frame problem: a mathematical investigation of the common sense law of inertia.* MIT Press, Cambridge, MA, 1997.

[34] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Practical Aspects of Declarative Languages*, pp. 305-319. Springer-Verlag, 1999.

[35] H. Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, Vol. 31, No. 1-3, pp. 245-298, 1997.

[36] J. F. Wakerly. *Digital Design Principles and Practices.* Prentice Hall, 1994.

[37] R. Watson. An application of action theory to the space shuttle. In G. Gupta, editor, *LNCS – Proc. of Practical Aspects of Declarative Languages '99*, Vol. 1551, pp. 290-304, 1999.