

Constraint Processing and Logic Programming *

James Bowen[†] and Dennis Bahler[‡]
Department of Computer Science,
North Carolina State University,
Raleigh, NC 27695-8206

Abstract

Constraint processing is complementary to logic programming: logic programming is concerned with proving theorems from a set Γ of sentences which are assumed to have a model; constraint processing is concerned with finding a model for Γ . This view of constraint processing has been realized in the language Galileo which we introduce in this paper. In Galileo, any sentence (atomic, compound or quantified) from full first-order classical logic is a well-formed constraint. In fact, classical logic sentences are only a subset of the well-formed constraints in Galileo; any sentence in first-order Free Logic, which subsumes classical logic, is a well-formed constraint in Galileo. This expressive power comes at the price of involving the user in the constraint satisfaction process. The user of a Galileo program and the Galileo interpreter form a composite constraint satisfaction system in which the run-time system performs what we call constraint *monitoring*. We have successfully used Galileo to implement several CAD applications, fragments from one of which we present in the paper, in order to illustrate the usage of Free Logic.

1 Introduction

There are several existing logic programming languages, all based essentially on the same treatment of problem-solving. In these systems, the task of solving a problem is treated as the task of proving a theorem ϕ given a set Γ of axioms. The well-formed formulae in Γ are treated as axioms because it is assumed there exists some interpretation \mathcal{I} which is a model of Γ , that is, $\exists \mathcal{I} \models_{\mathcal{I}} \Gamma$. Put another way, if the axioms have no model, they are not mutually consistent and conventional deduction provides no means of distinguishing valid theorems from invalid in that case.

Parsing is one well-known application of this problem-solving model. In this kind of application, a grammar and lexicon are presented in the form of a set of axioms, and an

*This work was supported in part by the National Science Foundation under grant DDM-8914200.

[†]jabowen@cscadm.ncsu.edu

[‡]drb@cscadm.ncsu.edu

utterance to be parsed is presented as a theorem to be proved about a set of terminal symbols. The underlying assumption in parsing is that there exists some interpretation for the constant symbols which renders the axioms true, i.e., which constitutes a legal parse.

The need to insure that a model exists for a set of axioms suggests another style of programming based on logic, one which is complementary to that underlying such theorem-proving languages as Prolog, Prolog II [6], Prolog III [4], CHIP [5], or CLP(\mathfrak{R}) [7]. In this complementary approach, problem-solving is not treated as the task of proving a theorem from a set of axioms. Instead, given axioms Γ and a universe of discourse \mathcal{U} , the questions of interest are whether Γ can be semantically modeled in \mathcal{U} and, if so, how the symbols appearing in Γ are interpreted.

In general, semantic modeling involves finding an interpretation for each object, function or relation symbol appearing in Γ that renders the sentences in Γ true. Constraint processing is semantic modeling of a restricted form: in the statement of a constraint problem, interpretations are provided for the function and relation symbols; interpretations need only be found for the object symbols.

In this view of constraint processing, constraint satisfaction becomes the following task: given

- a set Γ of sentences,
- a universe of discourse \mathcal{U} , and
- an interpretation for the function and relation symbols appearing in Γ ,

find an interpretation for the object symbols appearing in Γ that renders the sentences in Γ true.

This view of constraint processing has been realized in the language Galileo which we introduce in this paper. In Galileo, any sentence (atomic, compound or quantified) from full first-order classical logic is a well-formed constraint. In fact, classical logic sentences are only a subset of the well-formed constraints in Galileo; any sentence in first-order Free Logic, which subsumes classical logic, is a well-formed constraint in Galileo.

This expressive power comes at the price of involving the user in the constraint satisfaction process. The user of a Galileo program and the Galileo interpreter form a composite constraint satisfaction system in which the run-time system performs what we call constraint *monitoring*, the details of which are beyond the scope of this paper (but see [1]).

In Section 2, we give a brief pragmatic overview of the chief features of Galileo. In Section 3, we review the chief concepts of Free Logic. In Section 4, we show how a program in Galileo consists of a theory in Free Logic with a set of ancillary statements which define a universe of discourse and an interpretation, based on this universe, of the function and relation symbols appearing in the theory. We have successfully used Galileo to implement several CAD applications and, in Section 5, we present a fragment from one such application, to illustrate the real-world application of Free Logic. In Section 6, we present our conclusions.

2 A Brief Overview of Galileo

Galileo is an extension of an earlier language called Leo [1]. A program in Galileo consists of a set of declarative statements. These fall into two main groups: constraints and ancillary statements. Every program must contain constraint statements and one type of ancillary statement, object declarations. The following simple program contains three object declarations, (1) - (3), and five constraints, (4) - (8)

- (1) area : integer
- (2) length : integer
- (3) breadth : integer
- (4) area = length * breadth
- (5) not(length = breadth)
- (6) area =< 10 or 16 =< area and area =< 20
- (7) area < 10 implies length =< breadth + 1
- (8) all X : integer(X) and X > 5 implies even(X)

An object declaration has two parts, the object's name and its domain.¹ For example, (1) declares the existence of an object whose name is "area" and which ranges over the domain of integers.

Any sentence in first-order classical logic is a well-formed constraint; this includes atomic sentences, such as (4), negated sentences, such as (5), compound sentences, constructed using the usual logic connectives (and, or, implies, equiv), such as (6) and (7), and quantified sentences, such as (8).

All the objects in this simple program range over the integers, a predefined domain; the predicates used (=, =<, >, <, even) and the functions used (*, +) are also predefined. However, most real-world applications (Galileo and its predecessor Leo have been used to build CAD systems for electrical, mechanical and civil engineering [1, 2, 3]) cannot be handled using only built-in domains, functions and relations. So the language supports ancillary statements which define application-specific domains, functions and relations; these can all be defined either extensionally or intensionally. In addition, the language also supports declarations about the (ir)reflexivity, (a)symmetry and transitivity of relations, the invertibility of functions, and the closures/restrictions of functions or relations.

Consider, for example, the following simple program, where " ::= " may be read as "is defined to be":

- (9) domain hue ::= {red,blue,green}
- (10) domain small_number ::= {X : number(X) and 0 =< X and X =< 20}
- (11) domain form ::= {round,rectangular}
- (12) domain plate ::= (color : hue, area : small_number, shape : form)
- (13) relation comp(hue,hue) ::= {(red,blue), (red,green)}
- (14) relation compatible(hue,hue) ::=

¹An object declaration may also contain a third item, a long synonym, which serves only to improve the user interface of an application program and is of no interest here.

- (15) symmetric closure of comp(hue,hue)
relation approx_equal(small_number,small_number) ::==
{ (X,Y) : abs(X-Y) =< 0.1*average(X,Y) }
- (16) function average(number,number) - > number ::==
{ (X,Y) - > Z : Z = (X+Y)/2 }
- (17) function drkr(hue,hue) - > hue ::==
{ (red,blue) - > blue, (red,green) - > green,
(blue,green) - > green }
- (18) function darker(hue,hue) - > hue ::==
commutative closure of drkr(hue,hue) - > hue
- (19) kolor : hue
- (20) plate1 : plate
- (21) plate2 : plate
- (22) compatible(plate1.color,plate2.color)
- (23) approx_equal(plate1.area,plate2.area)
- (24) kolor = darker(plate1.color,plate2.color)
- (25) exists Y : plate(Y) and Y.area > 8
- (26) all Y : plate(Y) implies
(Y.shape=round implies
exists(Y.radius : small_number) and
Y.area=3.14159*Y.radius*Y.radius) and
(Y.shape=rectangular implies
exists(Y.length : small_number) and
exists(Y.breadth : small_number) and
Y.area = Y.length * Y.breadth) and
(Y.area > 10 equiv Y.color = red).

Statements (9) through (18) define concepts which are referenced in statements (19) through (26). In (19), object “kolor” is declared to range over a domain “hue”; this (scalar) domain is defined extensionally in (9). In (20) and (21), objects “plate1” and “plate2” are declared to range over the domain of “plates”. This domain is defined in (12) to be structured.

The constraint in (22) specifies that the colors of plate1 and plate2 must be “compatible”. This relation is declared in (14) to be the symmetric closure of a relation “comp” which is declared extensionally in (13). Constraint (23) specifies that both plates must have approximately equal areas. The meaning of approximately equal is defined intensionally in (15). This intensional definition refers to two functions; one of these, “abs”, or absolute value, is system-defined but the other, “average”, is defined intensionally in (16).

Constraint (24) is different from (22) and (23) in that it is uni-directional; “kolor” can derive a value from the functional expression on the right-hand side of the equality but neither plate can derive its color from “kolor”. The function “darker” used in (24) is declared in (18) to be the commutative closure of a function “drkr” which is defined extensionally in (17).

Constraint (25) specifies that there must be at least one plate having an area larger than 8.

Constraint (26) states that a round plate has a radius and that its area is π times the square of the radius; it also states that a rectangular plate has a length and a breadth and that its area is the product of its length and breadth; finally, this constraint also states that all, and only, plates having an area bigger than 10 must/can be red.

When constraint (26) states that a round plate has a radius and a rectangular one has a length and breadth, it does so by means of a form of existential specification which is crucially different from the classical form of existential quantification found in constraint (25). This form of existential specification is based on Free Logic and will be discussed further in Section 3.

If the constraints in this program were based solely on classical logic, (12) would define the domain of “plates” to be the cross-product of three scalar domains,

$$\text{plate} = \text{hue} \times \text{small_number} \times \text{form},$$

where the scalar domains “small_number” and “form” are defined intensionally in (10) and (11). However, because constraint (26) is based on Free Logic this is not the case. Constraint (26) shows that

$$\text{hue} \times \text{small_number} \times \text{form} \times \text{small_number} \subset \text{plate}$$

and

$$\text{hue} \times \text{small_number} \times \text{form} \times \text{small_number} \times \text{small_number} \subset \text{plate}.$$

In fact,

$$\begin{aligned} \text{plate} = \\ & \text{hue} \times \text{small_number} \times \text{form} \times \text{small_number} \cup \\ & \text{hue} \times \text{small_number} \times \text{form} \times \text{small_number} \times \text{small_number}. \end{aligned}$$

A Galileo constraint network is processed by interactive collaboration between the system and the user [1]. User assertions about objects in the world represented by the program are interleaved with inferences made by the system. We call the computer-inferencing part of this interactive process constraint *monitoring*, to distinguish it from fully automatic constraint *satisfaction*.

User assertions can refer to individual objects or groups of objects. For example, when interacting with the program in (1) - (8), the user could assert

$$(27) \quad \text{area} = 6$$

or, when interacting with the program in (9) - (26), he could assert

$$(28) \quad \text{all } (X,Y) : \text{plate}(X) \text{ and } \text{plate}(Y) \text{ implies } X.\text{area} <> Y.\text{area}$$

It is worth noting that, while (27) somewhat resembles an assignment statement, it isn't; it is actually a constraint between two first-class objects. As shown below, the only difference between “area” and “6” is that the latter object has a predefined interpretation.

3 Free Logic

Free logic is an extension of classical logic. In classical logic, a model \mathcal{M} consists of a universe \mathcal{U} and an interpretation function \mathcal{I} . The function \mathcal{I} assigns to each predicate constant p ² a relation $\mathcal{I}(p)$ among the elements of \mathcal{U} and some element of \mathcal{U} to every object constant. The model-theoretic semantics of classical logic may be expressed as follows:

- (a) $\mathcal{M} \models p(a_1, \dots, a_n)$ if and only if $(\mathcal{I}(a_1), \dots, \mathcal{I}(a_n))$ is in $\mathcal{I}(p)$.
- (b) $\mathcal{M} \models \neg A$ if and only if $(\mathcal{M} \not\models A)$.
- (c) $\mathcal{M} \models A \wedge B$ if and only if $\mathcal{M} \models A$ and $\mathcal{M} \models B$.
- (d) $\mathcal{M} \models A \vee B$ if and only if $\mathcal{M} \models A$ or $\mathcal{M} \models B$.
- (e) $\mathcal{M} \models A \Rightarrow B$ if and only if $\mathcal{M} \not\models A$ or $\mathcal{M} \models B$.
- (f) $\mathcal{M} \models A \Leftrightarrow B$ if and only if $(\mathcal{M} \not\models A) \text{ and } \mathcal{M} \not\models B$ or $(\mathcal{M} \models A \text{ and } \mathcal{M} \models B)$.
- (g) $\mathcal{M} \models (\forall X)A$ if and only if $\mathcal{M} \models (\kappa/X)A$ for every object constant κ .
- (h) $\mathcal{M} \models (\exists X)A$ if and only if there is some object constant κ for which $\mathcal{M} \models (\kappa/X)A$.

In Free Logic [8], by contrast, a model \mathcal{M} contains, in addition to a universe \mathcal{U} and an interpretation function \mathcal{I} , a third item, a story \mathcal{S} . The function \mathcal{I} assigns to each predicate constant p a relation $\mathcal{I}(p)$ among the elements of \mathcal{U} . Note that, unlike in classical logical semantics, not every object constant need be assigned an element of \mathcal{U} by \mathcal{I} .

The story \mathcal{S} is a set (possibly empty) of atomic sentences, each of which contains some object constant to which \mathcal{I} has assigned no element of \mathcal{U} .

In Free Logic, truth is established as follows:

- (a) $\mathcal{M} \models p(a_1, \dots, a_n)$ if and only if $p(a_1, \dots, a_n)$ is in \mathcal{S} or $(\mathcal{I}(a_1), \dots, \mathcal{I}(a_n))$ is in $\mathcal{I}(p)$.
- (b) $\mathcal{M} \models \neg A$ if and only if $\mathcal{M} \not\models A$.
- (c) $\mathcal{M} \models A \wedge B$ if and only if $\mathcal{M} \models A$ and $\mathcal{M} \models B$.
- (d) $\mathcal{M} \models A \vee B$ if and only if $\mathcal{M} \models A$ or $\mathcal{M} \models B$.
- (e) $\mathcal{M} \models A \Rightarrow B$ if and only if $\mathcal{M} \not\models A$ or $\mathcal{M} \models B$.
- (f) $\mathcal{M} \models A \Leftrightarrow B$ if and only if $(\mathcal{M} \not\models A \text{ and } \mathcal{M} \not\models B)$ or $(\mathcal{M} \models A \text{ and } \mathcal{M} \models B)$.
- (g) $\mathcal{M} \models (\forall X)A$ if and only if $\mathcal{M} \models (\kappa/X)A$ for every object constant κ to which \mathcal{I} has assigned an element of \mathcal{U} .
- (h) $\mathcal{M} \models (\exists X)A$ if and only if there is some object constant κ to which \mathcal{I} has assigned an element of \mathcal{U} for which $\mathcal{M} \models (\kappa/X)A$.

²We need not consider functions separately, since an n -ary function is equivalent to an $n+1$ -ary relation.

This new form of model has interesting consequences. For example, in Free Logic $p(a)$ could be true because $p(a)$ is in the story \mathcal{S} , so the following derivation does not hold in the logic:

$$\frac{p(a)}{(\exists X)p(X)}$$

It is important to note that in a Free Logic model for a set of sentences Γ not every object constant appearing in Γ need be assigned an element of \mathcal{U} by \mathcal{I} . In one notational convention for Free Logic, a form of existential specification, denoted by E , is used to talk about whether or not an object constant is assigned an element of \mathcal{U} by \mathcal{I} . This is used in sentences of the form $E\kappa$ where κ is an object constant appearing in Γ ; this sentence states that κ is assigned an element of \mathcal{U} by \mathcal{I} . $E\kappa$ can be read as “the object denoted by κ exists” or “ κ designates some element of the universe of discourse.” To handle sentences using this notation, another member is added to the set of Free Logic semantic truth rules (a)-(h):

- (i) $\mathcal{M} \models E\kappa$ if and only if κ is assigned an element of \mathcal{U} by \mathcal{I} .

Now, the following derivation holds:

$$\frac{Ea}{\frac{p(a)}{(\exists X)p(X)}}$$

It is important to distinguish between three types of situation. Only one of these situations may be true at any one time regarding a given object constant κ .

1. An object constant κ is known to designate some element of the universe of discourse; this, of course, is the case for every object constant appearing in a set of sentences in classical logic; it is also the case for some object constants appearing in a set of sentences in Free Logic.
2. An object constant κ is known not to designate any element of the universe of discourse; this, of course, is never the case for any object constant appearing in a set of sentences in classical logic; it may, however, be the case for some object constants appearing in a set of sentences in Free Logic.
3. It is not known whether or not an object constant κ designates an element of the universe of discourse; this is never the case for any object constant appearing in a set of sentences in classical logic but, however, it may be the case for some object constants, at some stage in an inference process involving a set of sentences in Free Logic.

A situation of this third type becomes a situation of the first or second type by the derivation, respectively, of a sentence of the form $E\kappa$ or $\neg E\kappa$. For example, a derivation like the following transforms a situation of the third type into one of the first type:

$$\frac{p(a) \Rightarrow Eb}{\frac{p(a)}{Eb}}$$

In some contexts, such as in the middle of processing a set of constraints Γ , where we are constructing the interpretation function \mathcal{I} for Γ , it may be known that some object constant κ does designate an element of \mathcal{U} while it is not yet known which element is designated. It is important to recognize that this is a situation of the first type enumerated above. In a situation of type one, we can know that κ is in the domain of the mapping function \mathcal{I} without yet knowing what element of the function's range, \mathcal{U} , κ is mapped onto by \mathcal{I} ; this is the case in constraint processing where our task is to construct the mapping function \mathcal{I} .

4 Galileo Programs as Free Logic Theories with ancillary definitions

A Galileo program is a theory Γ in Free Logic, accompanied by a set of ancillary definitions which specify the universe of discourse \mathcal{U} for Γ and those pairs from the model's interpretation function \mathcal{I} which map the function and relation symbols appearing in Γ .

Consider, for example, the simple program in lines (1) through (8) of Section 2. This is equivalent to the following theory in Free Logic:

- (1a) $Earea$
- (1b) $number(area)$
- (2a) $Elength$
- (2b) $number(length)$
- (3a) $Ebreadth$
- (3b) $number(breadth)$
- (4a) $area = length * breadth$
- (5a) $\neg(length = breadth)$
- (6a) $area = < 10 \vee 16 = < area \wedge area = < 20$
- (7a) $area < 10 \Rightarrow length = < breadth + 1$
- (8a) $(\forall X)(number(X) \wedge X > 5 \Rightarrow even(X))$

Each object declaration is equivalent to two sentences in Free Logic; one sentence declares that an object constant does designate some element of \mathcal{U} , while the other declares that the object constant satisfies some unary predicate. For example, (1) specifies that the object constant $area$ designates an element of \mathcal{U} ³ and also specifies that $area$ satisfies the unary predicate $number$.

In this case, since the program did not include any domain definitions, the Galileo run-time system supplies the default universe of discourse which consists of the real numbers \mathfrak{R} . In addition, the run-time system always supplies the following:

- The additional axiom $(\forall X)(number(X) \Rightarrow (EX))$
- Axioms of the form $number(n)$ for each number n
- Those pairs in the interpretation function \mathcal{I} which map from numeric object constants like 10 onto real numbers like 10.0.⁴

³It is not, of course, yet known which element of \mathcal{U} is designated by $area$.

⁴We will follow the typographical convention that elements of \mathcal{U} be written in typewriter font.

- Those pairs in the interpretation function which map from predicate and function symbols like *number*, $=$, $<$, $*$ and $+$ onto the appropriate relations over the appropriate domain(s).

To construct a model for the theory in (1a) through (8a), we need only determine those parts of the interpretation function \mathcal{I} that map from the object constants *area*, *length*, and *breadth* to elements of \mathfrak{R} .

Now, consider the program in (9) through (26) above. This is equivalent to a theory in Free Logic and a set of ancillary definitions which define the universe of discourse and parts of the interpretation function.

Consider, for example, the domain definition in (9), which is repeated here for convenience:

$$(9) \quad \text{domain hue} ::= \{\text{red, blue, green}\}.$$

Typically, an extensional scalar domain definition such as (9) provides six types of information to the Free Logic theory:

- It contributes a set of elements to the universe of discourse; in the example, three elements are added to the universe, so that now $\mathfrak{R} \cup \{\text{red, blue, green}\} \subseteq \mathcal{U}$.
- It introduces a set of object constants; in the example, *red*, *blue* and *green*.
- It contributes to the interpretation function a set of one-to-one mappings from these object constants to these elements of the universe of discourse; in the example, the following mappings are added to \mathcal{I} : $\text{red} \rightarrow \mathbf{red}$; $\text{blue} \rightarrow \mathbf{blue}$; $\text{green} \rightarrow \mathbf{green}$.
- It declares that these object constants do designate elements of the universe of discourse; in the example, these sentences are added: *Ered*, *Eblue*, *Egreen*.
- It introduces a unary predicate constant; in the example, the unary predicate constant *hue* is introduced.
- It contributes to the interpretation function a mapping from this predicate symbol to a unary relation defined over the universe of discourse. in the example, the following mapping is added to \mathcal{I} : $\text{hue} \rightarrow \{\mathbf{red, blue, green}\}$.

Other definitions contribute variously to the theory:

- Typically, an intensional scalar domain definition, such as (10), provides only two types of information: it introduces a unary predicate constant; it contributes to the interpretation function a mapping from this predicate symbol to a unary relation defined over the universe of discourse. Thus, for example, the domain definition in (10), which is repeated here for convenience

$$(10) \quad \text{domain small_number} ::= \\ \{X : \text{number}(X) \text{ and } 0 = < X \text{ and } X = < 20\}$$

introduces the unary predicate constant *small_number* and contributes to \mathcal{I} a mapping from this predicate constant to a subset of \mathfrak{R} .

- A structured domain definition, such as (12), contributes, intensionally, a set of tuples to the universe of discourse; it introduces, intensionally, a set of structured object constants; it contributes, intensionally, to the interpretation function \mathcal{I} , a set of mappings from constant symbols to the tuples it has contributed to the universe of discourse and to their elements; it declares, intensionally, that certain structured object constants do designate elements of the universe of discourse; it introduces a unary predicate constant; it contributes to the interpretation function a mapping from this predicate symbol to a relation defined over the universe of discourse.
- A relation definition, such as those in (13) through (15), introduces a predicate constant and it contributes to the interpretation function a mapping from this predicate constant to a relation defined over the universe of discourse.
- A function definition, such as those in (16) through (18), introduces a function constant and it contributes to the interpretation function a mapping from this function constant to a relation defined over the universe of discourse.

Like any other constraint, a constraint which uses the special Free Logic form of existential specification contributes a sentence to the theory. Thus, for example, constraint (26) introduces the following sentence in Free Logic:

$$\begin{aligned}
 (26a) \quad & (\forall Y)(plate(Y) \Rightarrow \\
 & \quad (Y.shape = round \Rightarrow (EY.radius)and \\
 & \quad \quad Y.area = 3.14159 * Y.radius * Y.radius)and \\
 & \quad (Y.shape = rectangular \Rightarrow \\
 & \quad \quad (EY.length)and(EY.breadth)and \\
 & \quad \quad Y.area = Y.length * Y.breadth)and \\
 & \quad (Y.area > 10 \Leftrightarrow Y.color = red)).
 \end{aligned}$$

In addition, as explained above, a constraint which uses the special Free Logic form of existential specification may interact with a structured domain definition to contribute to the universe of discourse and to the interpretation function.

5 Example Free Logic Application

We have built several CAD applications using Galileo [1, 2, 3]. Here we introduce a short fragment from an automated assistant for Design for Testability (DFT) which has been implemented in Galileo. We focus on this small part of the larger program to illustrate the utility in real-world applications of the Free Logic aspects of Galileo.

The following are some rules about the DFT of printed wiring boards provided by our domain experts:

- Each VLSI device of the following types on a board must have its own oscillating crystal on the board: cpu devices, communication controllers, and digital signal processors.

- If a crystal’s oscillation frequency exceeds the maximum oscillation frequency that can be handled by any of the pieces of test equipment that will be used to analyze the board, the board must also contain a divider circuit associated with each such crystal.
- Each crystal on a board must have its own compatible pullup resistor if the crystal uses on-chip disablement; otherwise, if it uses off-chip disablement, there must be a tristate buffer associated with the crystal.

Here we present, without the context of the overall program, the Galileo constraints which correspond to these requirements:

- all $X : \text{vlsi}(X)$ and $X.\text{type} \in \{\text{cpu}, \text{comm_cont}, \text{dsp}\}$
implies $\text{exists}(X.\text{crystal} : \text{crystal})$.
- all $X : \text{crystal}(X)$ implies
($\text{exists } Y : \text{tester}(Y)$ and $Y.\text{maxfreq} < X.\text{freq}$ implies
 $\text{exists}(X.\text{ancillary_circuit} : \text{seqdev})$ and
 $X.\text{ancillary_circuit.type} = \text{divider}$).
- all $X : \text{crystal}(X)$ implies
($X.\text{dis}=\text{onchip}$ implies
 $\text{exists}(X.\text{pullup} : \text{resistor})$ and
 $\text{compatible}(X.\text{pullup.watts}, X.\text{pullup.ohms})$) and
($X.\text{dis}=\text{offchip}$ implies
 $\text{exists}(X.\text{trstbuf} : \text{tristatbuffer})$ and
 $\text{compatible}(X.\text{trstbuf.pullup.watts}, X.\text{trstbuf.pullup.ohms})$).

6 Summary

Most logical systems attempt to prove a conclusion from a set of axioms that is assumed to be consistent, i.e., for which a model is presumed to exist. Constraint processing systems, by contrast, attempt to determine if a set of logical sentences (the constraints) has a model, that is, an interpretation in which all the constraints are simultaneously satisfied. Extending this logical view of constraint processing to Free Logic provides a rigorous mathematical foundation for the creation of constraint systems in which (i) any sentence in first-order logic is a legal constraint, and (ii) the set of objects is not constant, but instead contingent on existing objects and/or the values of those objects. These ideas have been implemented in the constraint language Galileo, which has been used to build several successful CAD applications.

References

- [1] Bowen J and O’Grady P, 1989. “A Constraint Programming Language for Life Cycle Engineering”, Technical Report, Center for Intelligent Systems in Design and Manufac-

turing, North Carolina State University. Submitted to *International Journal of Artificial Intelligence in Engineering*.

- [2] Bowen J and O'Grady P, 1990. "Life Cycle Design Advisors: Characteristics and Technology", Technical Report, Center for Intelligent Systems in Design and Manufacturing, North Carolina State University.
- [3] Bowen J and Elsaidy W, 1990. "Constraint-based Design of Elevator Hoistways", Technical Report, Center for Intelligent Systems in Design and Manufacturing, North Carolina State University.
- [4] Colmerauer A, 1987. "An Introduction to Prolog III". Draft, Groupe Intelligence Artificielle, Universite Aix-Marseille II.
- [5] Dincbas M, van Hentenryck P, Simonis HY, Aggoun A, Graf T and Berthier F, 1988. "The Constraint Logic Programming Language CHIP", in *Proceedings FGCS'88*.
- [6] Giannesini F, Kanoui H, Pasero R and van Caneghem M, 1986. *Prolog*, Reading, MA: Addison-Wesley.
- [7] Jaffar J and Michaylov S, 1986. "Methodology and Implementation of a CLP System", *Proc. 4th International Conference on Logic Programming*.
- [8] Lambert K and van Fraassen B, 1972. *Derivation and Counterexample: An Introduction to Philosophical Logic*, Enrico, CA: Dickenson Publishing Company.