PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

THE CONSTRAINT LOGIC PROGRAMMING LANGUAGE CHIP

M.Dincbas, P.Van Hentenryck, H.Simonis, A.Aggoun, T.Graf, F.Berthier

European Computer-Industry Research Centre (ECRC)
Arabellastr. 17, 8000 Muenchen 81, F.R.G.

ABSTRACT

CHIP is a new logic language combining the declarative aspects of logic programming with the efficiency of constraint solving techniques. It has been designed to tackle real world constrained search problems. It extends usual Prolog-like logic languages by introducing three new computation domains namely finite domain restricted terms, boolean terms and linear rational terms. For each of them CHIP uses specialized constraint solving techniques: consistency techniques for finite domains, equation solving in Boolean algebra and a symbolic simplex-like algorithm for rationals. CHIP has been successfully applied to a large number of industrial problems especially in the areas of planning, scheduling and circuit design. For all these problems the flexibility and the efficiency of CHIP has been systematically demonstrated.

1 INTRODUCTION

Many real life problems like scheduling, allocation, layout, fault diagnosis and hardware design verification can be seen as **constrained search** problems. Most of them belong to the class of NP-complete problems [16]. The most common approach for solving these problems consists in writing specialized programs in procedural languages. This approach requires substantial effort for program development, and the resulting programs are very hard to maintain, modify and extend. CHIP (Constraint Handling In Prolog) is an attempt to overcome these difficulties by providing, besides the efficiency of conventional approaches, the main features of fifth-generation tools: declarativeness and flexibility. It is a Prolog-like logic programming language extended by symbolic and numerical constraint solving techniques.

Logic programming is very appropriate to state constrained search problems. Its relational form and the logical variables are very adequate to formulate such problems in a declarative way and its nondeterministic computation liberates the user from the tree-search programming. However, usual logic languages like Prolog are too inefficient to tackle large search problems. CHIP differs from them by its active use of constraints in order to reduce the search space as much as possible. The adequacy of CHIP to solve industrial problems has been shown during the last two years on a very wide range of application domains (see section 7).

Given a computation domain, a **constraint** expresses a relationship between some objects of this domain. Constraint manipulation and propagation have been studied in the Artificial Intelligence community in the late 70's and early 80's especially in the USA. They provide very interesting problem solving techniques like local value propagation, data driven computation, sophisticated search algorithms (e.g. forward checking) and consistency checking [13] [14] [18] [28] [30] [38] [39] [46]. The general idea behind these techniques is the use of constraints to prune the search space in an 'a priori' way, i.e. before the generation of values. Some problem solvers (like ALICE [27]) were developed using these techniques. But they were implemented as 'black-boxes' and suffered from the lack of flexibility.

In the framework of logic and theorem proving, G.Huet was first to introduce the concept of **constraint** in the early 70's. While working on the mechanization of higher-order logic, he replaced the problem of unification in the typed λ-calculus by the solving of equality constraints. He was talking about **constrained resolution** and **constrained clauses** [22].

The idea of introducing constraint solving techniques in Logic Programming is rather recent. However extensive work has been conducted in this direction during the last three years especially in Europe (Marseille, ECRC), USA (IBM Yorktown Heights), Canada (Vancouver) and Australia (Monash Univ.). The general idea behind the introduction of these extensions inside logic programming is the use of some mathematical tools (like simplex) to solve numerical constraints and the use of consistency checking and constraint propagation techniques to solve symbolic constraints. Apart from CHIP, there are mainly three systems which are currently in development: Prolog III, CLP($\Re$) and Trilogy. Prolog III has been developed by Colmeraurer and his team at the University of Marseille [5]. It uses a simplex-like algorithm to solve linear equations and inequalities on rational numbers for linear programming purposes. It also provides a saturation method to deal with boolean terms. The work on CLP conducted by J.Jaffar and J.L.Lassez at IBM-Yorktown Heights in collaboration with the University of Monash in Australia aims at giving a formal framework for a logic programming scheme based on constraint solving [23]. An instance of this scheme for handling linear equations and inequalities over real numbers has been implemented in CLP($\Re$) [24] [19]. Trilogy has been developed by P.Voda and his team at the University of Vancouver [45]. Unlike CHIP, Prolog III and CLP($\Re$), it does not provide the full power of Prolog. Indeed Trilogy is based on the Voda's own 'theory of pairs'. From the constraint point of view, it provides a decision procedure for the Presburger arithmetic.

In this paper we give an overview of CHIP. The presentation will be rather informal. We first give the motivations behind CHIP showing the necessity to introduce new computation domains other than the usual Herbrand universe. For each of the new computation domains (i.e. finite domains, booleans and rationals), we present the types of terms and constraints manipulated by CHIP, and the constraint solving techniques used for them. In the final section we give a panorama of applications which have already been solved in CHIP.

2 COMPUTATION DOMAINS OF CHIP

CHIP is a new logic language combining the declarative aspect of logic programming with the efficiency of constraint solving techniques. It is based on the concept of **active** use of constraints [15] [43] [9]. It differs from usual logic languages in two aspects: the computation domains on which it works, and the constraint handling and improved search procedures it provides.

The power of a Prolog-like logic programming language rests on three mechanisms: relational form, unification and non-deterministic computation. Prolog carries out computations in the Herbrand universe. Unification is used to solve equations in this universe, i.e. on **uninterpreted** terms. Therefore when modeling a problem, one has to use a mapping from its intended domain on the Herbrand universe. This causes the loss of not only the naturalness of the problem expression but also the efficiency of its resolution. We can provide inside Prolog richer computation domains than the Herbrand universe, and handle more expressive terms than the uninterpreted Herbrand terms. This entails to extend unification in Prolog in order to take into account the intended interpretations given to some functional symbols. On the other hand, unification is used only to solve equations i.e. just one kind of constraint. Other kinds of constraints like disequations (~=), inequalities on numbers (<, >=, etc.) or range restriction for variables can be imagined [10]. In Prolog these constraints are used in a **passive** way through the well-known **generate & test** paradigm which causes its legendary inefficiency on such problems. Another extension is then the introduction of more general constraint solving techniques developed in Artificial Intelligence, Mathematics and Operations Research for these specialized computation domains.

The question that arises at this point is which computation domains should be introduced into logic programming with their corresponding constraint solving techniques. Three major criteria for this choice are: the expressive power of the computation domain, the existence of efficient constraint solving techniques for it and its interest for possible applications. Following these criteria, we have chosen the following three additional computation domains for CHIP:

- Finite domain restrictive terms
- Boolean terms
- Linear rational terms

From our experience of studying and solving problems in a wide range of application domains, we can say that these extensions are enough to cover a large variety of interesting problems. For each of these new computation domains, CHIP uses specialized algorithms. While consistency checking techniques are used for finite domains, more mathematical tools (equation solving in boolean algebra and a symbolic simplex-like algorithm) are used for booleans and rationals. More details about these techniques will be given in the next sections. We can note that these extensions do not change the declarative aspects of logic programming.

3 FINITE DOMAINS

A large variety of problems can be viewed as discrete combinatorial problems. Their general form can be described as the search, in a discrete finite space, for a particular point satisfying a given set of constraints. In the present section, we discuss some features of CHIP devised to solve efficiently problems from this class.

3.1 Domain Variables

The basic feature of CHIP for solving discrete combinatorial problems is the ability to work on domain-variables, i.e. variables ranging over a finite domain [43]. CHIP differentiates between two different kinds of such variables, those ranging over constants, and those ranging over a finite set of natural numbers.

CHIP has also the ability to cope with arithmetic terms over domain-variables. These terms are constructed from natural numbers, domain-variables over natural numbers and the operators +, -, × and /.

3.2 Constraints over Finite Domains

CHIP provides a large variety of constraints on domain-variables. It contains not only **arithmetic** but also **symbolic** and even **user-defined** constraints. In this subsection, we give a flavor of the constraints on domain-variables available in CHIP.

3.2.1 Arithmetic constraints

As far as arithmetic constraints are concerned, CHIP allows the usual relations on arithmetic terms over domain-variables. For instance, for any such terms `X` and `Y`,

```
X > Y, X >= Y, X < Y, X =< Y, X = Y, X ~= Y
```

are well-formed constraints of CHIP.

3.2.2 Symbolic constraints

CHIP is not restricted to arithmetic constraints. It also contains, and this is part of its originality, symbolic constraints on domain-variables. Examples of some symbolic constraints (besides `X ~= Y`) are

- `element(Nb,List,Var)` which holds if `Var` is the `Nb`th element of `List`; this constraint can be used when `Nb` and `Var` are domain-variables or constants, and `List` is a list of domain-variables or constants.

- `alldistinct(List)` which holds if all elements of the list `List` are different; this constraint can be used if `List` is a list whose elements are either domain-variables or constants.

Constraints of these kinds are essential tools for solving many discrete combinatorial problems. For instance the `element` constraint can be used to enforce a relationship between two variables. They often allow more natural problem statements and more efficient problem-solving.

3.2.3 User-defined constraints

Not all constraints can be provided as primitives in a constraint language. It is therefore important to allow the programmer to define his/her own constraints. In CHIP it is possible to specify that a particular predicate is to be handled as a constraint using consistency techniques. The only requirement for a predicate to be handled as a constraint is that its ground instances either succeed or finitely fails (see below for an example). To our knowledge CHIP is the only constraint logic language which allows user-defined constraints.

3.2.4 Higher-order extensions

CHIP also includes some higher-order extensions for finding solutions optimizing (i.e. minimizing or maximizing) some evaluation function. These predicates can be used for solving combinatorial optimization problems. For instance, the predicate

```
minimize(Goal,Function)
```

where `Goal` is a predicate and `Function` is an arithmetic term over domain-variables can be used to find the solution of `Goal` which minimizes `Function`. It is implemented using a **branch & bound** technique.

3.3 Consistency Techniques

The previous subsections have discussed domain variables and constraints on domain variables. We now turn attention to the way these constraints are solved. All constraints involving domain-variables are solved through consistency techniques, a powerful paradigm emerging from AI to solve discrete combinatorial problems (e.g. [30] [28] [18]). The principle behind these techniques is to use constraints to reduce the domains of variables and thus the size of the search space. Different kinds of pruning (i.e. reduction of the domains) have been identified and efficient ways to achieve them have been devised. However, consistency techniques are usually not able to solve the constraints by their own. It follows that solving a discrete combinatorial problem using them consists in iterating the following two steps

- propagating the constraints as far as possible
- making a choice

until a solution is reached. It is also worth mentioning that constraints solved through consistency techniques are scheduled in a data-driven way.

A computational framework to embed consistency techniques inside logic programming has been defined in [41] [42]. It consists in three inference rules, the **Forward Checking Inference Rule** (the FCIR), the **Looking Ahead Inference Rule** (the LAIR) and the **Partial Looking Ahead Inference Rule** (the PLAIR), each one defining a particular way of pruning.

All primitive constraints on domain-variables in CHIP can be viewed as efficient specialization of these inference rules. For instance, `X ~= Y` (X different from Y) can be seen as a specialization of the FCIR. Other arithmetic relations on arithmetic terms over domain-variables can be seen as specializations of the PLAIR by means of a reasoning on variation intervals. For instance, given the constraint,

```
R + E + 1 = 10 + T
```

with R $\in \{0,1\}$ and E and T $\in \{0,2,3,4,5,6,7,8,9\}$ it will be inferred that T = 0 and E $\in \{8,9\}$.

The inference rules are also the basis of general **control mechanisms** for handling user-defined constraints. For instance, **forward** declarations [44] and **lookahead** declarations [42] make possible to use respectively the FCIR and the LAIR for any user-defined constraint. The following example give the use of a lookahead declaration to define a constraint for a junction of type 'arrow' in 3-dimensional scene analysis (d stands for domain-variable).

```
?- lookahead arrow(d,d,d).

arrow(convex,concave,convex).
arrow(concave,convex,concave).
arrow(arrow-in,convex,arrow-out).
```

3.4 Discussion

Domain-variables are a significant extension to logic programming and consistency techniques provide a uniform and efficient paradigm for solving constraints on finite domains, whether the constraints are arithmetic, symbolic or user-defined. The finite domain extension is used for constraint satisfaction, integer programming and combinatorial optimization problems. Many large problems have been solved within CHIP with an efficiency comparable to programs written in procedural languages, showing the adequacy of these techniques. See section 7 for more details about these applications.

Of course, for some arithmetic constraints, other techniques could have been considered, for instance a decision procedure for the Presburger arithmetic as done in Trilogy [45] or diophantine equation-solving methods [21]. However these mathematical tools were felt computationally too expensive and not adapted to solve the real world problems we are looking at. In addition they only apply to constraints on linear arithmetic terms and they do not exploit the main feature of the above class of problems: the finiteness of domains.

4 BOOLEAN UNIFICATION

In this section we want to explain the implementation and the use of boolean unification inside the CHIP system. It can be used for many problems in digital circuit design (verification, synthesis, simplification, test generation) and as a theorem-prover for propositional calculus.

Boolean unification solves symbolically equations over boolean terms. It forms a unitary theory [34] i.e. there is at most one most general unifier (mgu) for every equation. Different unification algorithms for boolean unification have been proposed [4] [29]. In our implementation we use the variable elimination algorithm of [4].

4.1 Boolean Terms

Since boolean unification provides a decision procedure for propositional calculus and is therefore NP-complete [16], the abovementioned algorithms all have an exponential worst case complexity. It is thus very important to use a compact description of boolean terms to achieve efficiency. Normal forms like DNF or sum-of-products require exponential space for the representation of many interesting functions.

In our implementation we distinguish between the external and the internal representation of boolean terms. In the external representation, boolean terms are built from truth values (0 and 1), from constants (atoms), from variables and from the boolean operators & (and), ! (or), # (xor), nand, nor, not. Constants usually denote symbolic input values, variables denote intermediate or output values. Internally, boolean terms are represented as directed acyclic graphs (dags). We use routines similar to the ones described in [3] to simplify terms and to perform boolean operations. The unification algorithm has been extended to take advantage of this efficient internal representation.

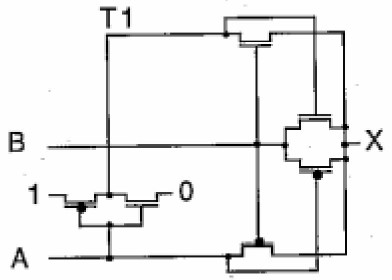
4.2 Use of Boolean Unification

In order to tell the system for which arguments we want to use boolean unification, we have to declare **bool-declarations** for predicates. Stating a declaration

```
?-declare and(h,h,bool,bool,bool).
and(M,N,X,Y,X&Y).
```

for the predicate and, we use normal (Robinson's) unification on Herbrand terms [33] for the first two arguments and boolean unification for the last three arguments. The predicate states that the last argument must be equal to the logical-and of the third and fourth argument.

Figure 1: Xor-gate Circuit Diagram



As a simple example for the use of boolean unification in CHIP take the verification of an xor-gate (see figure 1). The circuit description is explained in more detail in [35][37]:

```
?-declare eq(bool,bool).
eq(X,X).

?-declare n_switch(bool,bool,bool).
n_switch(Drain,Gate,Source):-
        eq(Drain&Gate,Gate&Source).

?-declare p_switch(bool,bool,bool).
p_switch(Drain,Gate,Source):-
        eq(Drain&not(Gate),
           not(Gate)&Source).

?-declare xor(bool,bool,bool).
xor(A,B,X):-
        p_switch(1,A,T1),
        n_switch(0,A,T1),
        p_switch(B,A,X),
        n_switch(B,T1,X),
        p_switch(A,B,X),
        n_switch(T1,B,X).
```

If we want to compute symbolically the output of this circuit, we have to solve the boolean equation of each switch. This instantiates the variables X and T1 to some boolean terms. We show the values of X and T1 after each step. Variables beginning with an underscore are free variables introduced by boolean unification.

```
?- xor(a,b,X).

1)  T1 =  1 # a #  _A&a
2)  T1 =  1 # a
3)  X  =  b # _C&a # a&b
4)  X  =  b # _C&a # a&b
5)  X  =  a # b # _D&a&b
6)  X  =  a # b
```

```
X = a # b
```

At the end of the computation, we see the value X = a # b as a result.

4.3 Discussion

Other systems [47] [20] use rewriting techniques to simplify symbolic boolean equations. Terms are rewritten into a sum-of-product normal form. These normal forms often require exponential space, therefore they can be used only to describe rather small problems. In Prolog III [5], a saturation method is used to solve boolean equations. This method does not compute a most general solution. Problems like circuit verification cannot be solved with this method. There exist several specialized programs for boolean term simplification for circuit design purposes [3] [1]. CHIP is comparable in efficiency to these systems, but provides a more declarative framework.

5 RATIONAL ARITHMETIC

We now turn to the part of CHIP which handles continuous problems, i.e. problems where there is an infinite number of points in the search space to explore, by modelling them through rational numbers.

5.1 Rational Terms

Rational terms are linear terms over rational numbers and rational variables (i.e. variables which take their values in rational numbers).

5.2 Constraints on Rational Terms

Usual arithmetic relations can be defined over rational terms. Given two rational terms X and Y, the following constraints

```
X > Y, X >= Y, X < Y, X =< Y, X = Y, X ~= Y
```

are all well-formed constraints of CHIP. Because of the restriction imposed on rational terms, only linear constraints are possible.

5.3 Solving Constraints on Linear Rational Terms

We now turn to the way constraints over rational terms are solved in our system. CHIP provides a decision procedure for constraints on linear rational terms. This means that for any set of constraints (over rational terms) CHIP can decide if they are satisfiable or not. This procedure is an adaptation of the simplex algorithm [6] [17] based on variable elimination and not on matrix manipulation. Given a set of constraints, the procedure will either fail (the constraints are not satisfiable) or will produce a set of bindings for the variables and a set of constraints in a simplified normal form.

From an implementation point of view, this procedure has a certain number of desirable properties:

Full integration: the constraint solver is fully embedded in the CHIP language and is not a separate module to which constraints are transmitted and results are fetched.

No fixed variable: the algorithm guarantees that all variables appearing in rational terms can take an infinite number of values. If a variable can only take one value (in this case we say that the variable is fixed), the procedure directly assigns this value to the variable. This property allows in particular to decide efficiently constraints of the form `X ~= Y`.

Incrementality: the procedure is incremental. If we have already solved a set S of constraints, adding a new constraint C to S will not require solving from scratch the set $S \bigcup \{C\}$. CHIP will transform the solution of S into a solution of $S \bigcup \{C\}$, if it exits. This property is of great importance since constraints in CHIP are created dynamically.

Uniformity: equations and inequalities are solved in a uniform manner by the introduction of **slack** variables. Contrary to the simplex algorithm, there is no need to retransform equalities into inequalities by introducing **artificial** variables.

From the user point of view, the inclusion of this procedure inside CHIP has a certain number of attractive properties:

Symbolic solutions: the solutions returned by the system are always most general ones. Hence CHIP can represent an infinite number of solutions in a finite way.

Strict inequalities: the user can express his/her problem not only in terms of inequalities but also in terms of **strict** inequalities (e.g. `X > Y`). This additional expressive power comes directly from the ability of CHIP to decide constraints of the form `X ~= Y`.

Satisfiability and optimization: CHIP can be used both for deciding if a set of constraints is satisfiable and for finding the most general solution to a set of constraints which optimizes (i.e. minimizes or maximizes) a linear evaluation function.

5.4 Discussion

The introduction of rational terms in CHIP offers a variety of techniques ranging from simple equation-solving to Linear Programming. Many problems from fields like Operations Research and electrical circuit analysis fall into this framework, opening new applications areas for logic programming. In the following, we discuss some basic design choices concerning the rational arithmetic part of CHIP.

5.4.1 Simplex versus polynomial algorithms

One may wonder why CHIP uses a simplex-based algorithm and not the (newly discovered) algorithms of Khachian [26] and Karmarkar [25]. These more recent algorithms have of course the interesting property to be of polynomial complexity in the worst case contrary to the simplex algorithm. However experimental studies have shown that simplex behaves very well in the average case (it is quasi linear), reinforcing its position as one of the most important tool in Operations Research. Khachian's algorithm, which was the first polynomial algorithm for Linear Progarmming, induces an unacceptable overhead. Karmarkar's algorithm seems to have great potential but it is not yet clear how to make it incremental.

5.4.2 Rational versus real numbers

In CHIP we have chosen to include rational arithmetic as in PROLOG III [5] and not real arithmetic represented by floating point numbers as in CLP($\Re$) [24]. This choice has been made because, for linear constraints, rational arithmetic has the desirable property of numerical stability. Rational numbers can be represented exactly on a computer whereas real numbers are normally represented by floating point numbers introducing rounding errors. It follows that working with rational numbers preserves the **soundness** of the system. This is not true if one works with floating point numbers. The system might well answer **yes** to a query when it is not a logical consequence of the program and **no** if it is. This is especially likely to occur if redundant or disequality constraints are involved in the problem. Of course, working with rational numbers induces some drawbacks in terms of time efficiency and memory consumption. It is necessary to use infinite precision numbers and to redefine the common arithmetic operations.

5.4.3 Linear versus non-linear terms

Non-linear constraints have not been included in CHIP. The reason is that no general analytical method is available for solving them. Iterative numerical methods will be necessary in most cases. Hence they suffer from numerical instability and thus they do not fit very well to the logic programming philosophy.

6 DEMON CONSTRUCTS

Several of the abovementioned techniques make use of a demon-driven computation. In the present section, we present some additional demon constructs which can be used for user-defined constraints, namely delay declarations, local and conditional propagation.

6.1 Delay Declaration

CHIP contains a delay mechanism, the delay declarations, which enables coroutining in a demon-driven way. For instance, the following program defines a storage element of type latch used in simulation of sequential circuits using a delay declaration. The meaning of this definition is to delay the call of the predicate `latch` until the first argument is nonvar.

```
?- delay latch(nonvar,any,any,any,any).
latch([],[],[],[],_).
latch([CLK|CLK1],[Ld|Ld1],[D|D1],[Q|Q1],S):-
        la1(CLK,Ld,D,Q,S),
        latch(CLK1,Ld1,D1,Q1,Q).

?- delay la1(nonvar,nonvar,any,any,any).
la1(CLK,1,D,D,S).
la1(CLK,0,D,S,S).
```

6.2 Local Propagation

In local propagation, constraints are used to find the values of uninstantiated variables from instantiated variables. This technique has been widely used in AI; Sussman and Steele's CONSTRAINTS language [39] is based on this computational paradigm while many algorithms in the area of hardware design and graphical systems are based on similar principles [2] [8] [7]. Local propagation can be implemented through a delay mechanism. However, this solution is neither elegant nor efficient. GHC [40] could also be used for implementing local propagation.

In CHIP local propagation is achieved through a general mechanism called **demon declaration**. The following program illustrates its use for a logical and gate.

```
?- demon and/3.
and(0,Y,Z) :- Z = 0.
and(Y,0,Z) :- Z = 0.
and(1,Y,Z) :- Y = Z.
and(X,1,Z) :- Y = Z.
and(Y,Y,Z) :- Y = Z.
and(Y,Z,1) :- Y = 1, Z = 1.
and(Y,Z,0) :- one_zero(Y,Z).

?- demon one_zero/2.
one_zero(0,_).
one_zero(_,0).
one_zero(X,1) :- X = 0.
one_zero(1,X) :- X = 0.
```

Predicates submitted to a demon declaration are used as rewrite rules. They are deterministic and a goal can only be resolved against them when it matches (in the sense of one-way unification) a head in one of the definition clauses. For instance the goal $\leftarrow$ `and(1,0,Z)` can be resolved against the second or third clause. One and only one of these clauses will be selected and the goal will be rewritten as `Z = 0`. Contrariwise the goal $\leftarrow$ `and(X,Y,Y)` does not match any head and thus is delayed until further information is available.

Demons have multiple applications ranging from qualitative reasoning to diagnostic from first principles [36] [35].

6.3 Conditional Propagation

Conditional propagation is a propagation technique driven by the satisfiability or unsatisfiability of constraints. Declaratively, it is a simple **if_then_else** construct.

```
if CONDITION then GOAL-1 else GOAL-2
```

Procedurally, it provides an efficient demon-driven mechanism. It is an extension of the **if_then_else** of Mu-Prolog [31] and is handled in the following way. For any allowed condition, CHIP has at his disposal a procedure able to decide if the condition is always true or always false for all instances of the condition or if the condition is true for some instances and false for some others. Therefore facing such a constraint, CHIP uses the adequate procedure to evaluate the `CONDITION`. If it always evaluates to true, `GOAL-1` is executed. If it always evaluates to false, `GOAL-2` is executed. Otherwise the `if_then_else` construct delays, waiting for more information. Any constraint on domain-variables, boolean terms and rational terms can make up an allowed condition.

Conditional propagation has been an important tool for the simulation of industrial electro-mechanical circuits and applications in qualitative reasoning.

7 APPLICATIONS OF CHIP

In this section we describe very briefly different real world problems solved in CHIP. Some of them were previously solved in conventional languages requiring a long development time and effort. CHIP drastically reduces this time while achieving a similar efficiency. The wealth of applications show the flexibility of CHIP to adapt to different problem areas. More about some of these problems can be found in [36] [35] [37] [42] [11] [12].

7.1 Applications in Scheduling and Planning

Operations Research (O.R.) is an inexhaustible source of interesting search problems. Many O.R. problems have been solved in CHIP, especially large-scale scheduling and planning problems.

Disjunctive scheduling We have applied CHIP to solve a disjunctive scheduling problem in civil engineering. The problem is to minimize the total duration of building a bridge with precedence and disjunctive constraints due to the limited availability of resources.

Graph colouring The problem is to find the minimum number of colours to label the vertices of a graph such that no two adjacent vertices are assigned to the same colour. Graphs containing one hundred nodes and thousands of vertices are efficiently colored with CHIP.

Car sequencing This problem occurs in the scheduling for the assembly line of car manufacturing. Each car may require a different set of options and the assembly line has capacity constraints for the options. The problem is to generate a sequence of cars which satisfies the capacity constraints.

Optimal traffic assignment for satellites This problem concerns the scheduling of the on-board switching systems in telecommunication satellites. The problem can be formulated as follows: given an interstation traffic matrix, determine the successive switching modes in order to switch all the traffic requirements in minimum time.

Warehouse location The problem is, given a set of possible warehouse locations, a set of customers, and the costs of stocking and transportation from warehouses to customers, to find an optimal configuration of the warehouses (i.e. their number and locations) which minimizes the total cost.

Cutting problems The problem consists in cutting (two dimensional) shelves of various sizes according to customer requirements from standard wood boards in a furniture factory. The objective is to minimize the total waste.

Investment planning The program chooses among different investment types in order to minimize or maximize a goal function over a given period. Thanks to the symbolic simplex method, the program yields the most general solution and the user can then interact with it to get the best solution with regards to his need.

Macro economics model The problem is the qualitative analysis of a macro economics model. Relations between factors in the models are given by a three valued (increasing, decreasing, constant) system.

7.2 Applications in Circuit Design

Another very promising application domain for CHIP is Circuit Design. The extensions provided in CHIP make it possible to solve large classes of problems for big circuits.

Circuit simulation The demon and constraint propagation mechanisms of CHIP allow the simulation of large combinatorial and sequential circuits (with several thousands of components).

Symbolic verification Symbolic verification means the formal comparison of an implementation of the circuit with its functional specification (usually a set of boolean equations).

Circuit synthesis For combinatorial circuits, we have developed in CHIP a program which automatically generates a circuit at the transistor level (in different technologies) from a truth-table or boolean equation specification.

Fault diagnosis The problem is to locate a faulty component in a circuit from its input/output misbehaviour. We make a single fault assumption, the diagnosis is based on model reasoning. The fault finding process uses a constraint relaxation method combined with a consistent labeling technique.

Automatic test-pattern generation We want to generate a minimal number of test patterns to detect all single stuck-at errors in combinational circuits. The CHIP program uses consistent labeling techniques with fault insertion to generate test patterns.

Circuit specialization and simplification The problem is, from the description of a circuit which performs a set of functions, to derive a more specialized one performing only a subset of the functions. The goal is to minimize the number of components in the simplified circuit.

Channel routing This application comes from the area of VLSI layout design. It consists in connecting terminals on two sides of a rectangular channel in presence of certain constraints. The objective is to minimize the channel width.

Microcode label assignment The microcode label assignment problem comes from the area of computer firmware development. The problem is to assign labels of symbolic microcode to addresses in a page of microcode memory. Branch instructions generate constraints on certain bit patterns.

Simulation of electro-mechanical devices For the analysis of hybrid circuits, a quantitative simulator using analog device models has been developed in CHIP. The system has been successfully applied to real world circuits coming from aerospace industry.

8 CONCLUSION

In this paper we have given an overview of the constraint logic programming language CHIP. It extends usual Prolog-like logic languages by introducing three new computation domains, finite domain restricted terms, boolean terms and linear rational terms. For each of them CHIP uses specialized constraint solving techniques: consistency techniques for finite domains, equation solving in Boolean algebra and a symbolic simplex-like algorithm for rationals. These extensions are combined with a demon-driven computation mechanism. Since these techniques are embedded in a programming language, heuristics specific to different problems can be easily added when necessary.

CHIP is very well-suited to solve especially constrained search problems. Keeping the main features of fifth-generation tools, i.e. declarativeness and flexibility, it brings into logic programming the efficiency of special purpose programs written in imperative languages. Several real life problems in the area of scheduling, planning and circuit design have been already solved in CHIP as efficiently as with a conventional approach but with much less programming effort and providing more flexibility in the solution. Among these applications we will mention just two: the so-called car-sequencing problem and the formal verification of hardware. The first problem has been recently presented as a challenge for AI technology [32]. The CHIP formulation of the problem is very elegant and large instances have been efficiently solved [12]. On the other hand, standard examples of ALU's (from 8-bit to 64-bit) and a small 16-bit microprocessor have been formally verified in CHIP [37].

ACKNOWLEDGMENTS

We gratefully thank Herve Gallaire for his continuous support and encouragements and Alexander Herold for fruitful discussions.

REFERENCES

[1] J.P. Billon and J.C. Madre. Original Concepts of PRIAM, an Industrial Tool for Efficient Formal Verification of Combinational Circuits. In *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988.

[2] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.

[3] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[5] A. Colmerauer. Opening the Prolog-III Universe. *BYTE Magazine*, 12(9), August 1987.

[6] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, New Jersey, 1963.

[7] R. Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24():347–410, 1984.

[8] J. de Kleer. *Local Methods of Localizing Faults in Electronic Circuits.* Technical Report AIM-394, Artificial Intelligence Laboratory, MIT, Cambridge, USA, 1976.

[9] M. Dincbas. Constraints, Logic Programming and Deductive Databases. In *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium*, pages 1–27, ICOT, Tokyo, Japan, October 1986.

[10] M. Dincbas, H. Simonis, and P. Van Hentenryck. Extending Equation Solving and Constraint Handling in Logic Programming. In *Colloquium on Resolution of Equations in Algebraic Structures (CREAS)*, Texas, May 1987.

[11] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming* (To appear).

[12] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, Munich, W.Germany, August 1988.

[13] M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Technical Report CMU-CS-83-161, Carnegie-Mellon University, December 1983.

[14] E.C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21:958–966, November 1978.

[15] H. Gallaire. Logic Programming: Further Developments. In *IEEE Symposium on Logic Programming*, pages 88–99, Boston, July 1985. Invited paper.

[16] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, New York, 1979.

[17] S.I. Gass. *Linear Programming.* McGraw-Hill, New York, 1985.

[18] R.M. Haralick and G.L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial intelligence*, 14:263–313, 1980.

[19] N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and Some Electrical Engineering Problems. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[20] J. Hsiang. Refutational Theorem Proving using Term-Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.

[21] G. Huet. An Algorithm to Generate the Basis of Solutions to Homogeneous Linear Diophantine Equations. *Information Processing Letters*, 7:144–147, 1978.

[22] G. Huet. *Constrained Resolution : A Complete Method for Higher Order Logic.* PhD thesis, Case Western Reserve University, August 1972.

[23] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich (FRG), January 1987.

[24] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[25] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4(4):373–395, 1984.

[26] L.G. Khachian. A Polynomial Algorithm in Linear Programming. *Soviet Math. Dokl.*, 20(1):191–194, 1979.

[27] J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[28] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[29] U. Martin and T. Nipkow. Unification in Boolean Rings. In *Proceedings of the 8th Conference on Automated Deduction*, pages 506–513, Oxford, England, July 1986.

[30] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132, 1974.

[31] L. Naish. *Mu-Prolog 3.1db Reference Manual.* Melbourne University Edition, 1984.

[32] B.D. Parrello. CAR WARS: The (Almost) Birth of an Expert System. *AI Expert*, 3(1):60–64, January 1988.

[33] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of ACM*, 12(1):23–41, January 1965.

[34] J.H. Siekmann. Universal unification. In *7th International Conference on Automated Deduction*, pages 1–42, Napa Valley, CA, 1984.

[35] H. Simonis and M. Dincbas. Using an Extended Prolog for Digital Circuit Design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, W.Germany, October 1987.

[36] H. Simonis and M. Dincbas. Using Logic Programming for Fault Diagnosis in Digital Circuits. In *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139–148, Geseke, W.Germany, September 1987.

[37] H. Simonis, H.N. Nguyen, and M. Dincbas. Verification of Digital Circuits Using CHIP. In *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988.

[38] G.L. Steele. *The Definition and Implementation of a Computer Programming Language based on Constraints.* PhD thesis, MIT, August 1980.

[39] G.J. Sussman and G.L. Steele. CONSTRAINTS: A Language for Expressing Almost-Hierarchical Descriptions. *AI Journal*, 14(1), 1980.

[40] K. Ueda. *Guarded Horn Clauses.* ICOT Tech. Report TR-103, Institute of New Generation Computer Technology, Tokyo, 1985.

[41] P. Van Hentenryck. A Theoretical Framework for Consistency Techniques in Logic Programming. In *IJCAI-87*, pages 2–8, Milan, Italy, August 1987.

[42] P. Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur (Belgium), July 1987.

[43] P. Van Hentenryck and M. Dincbas. Domains in Logic Programming. In *AAAI-86*, pages 759–765, Philadelphia, USA, August 1986.

[44] P. Van Hentenryck and M. Dincbas. Forward Checking in Logic Programming. In *Fourth International Conference on Logic Programming*, pages 229–256. Melbourne, Australia, May 1987.

[45] P. Voda. *The Constraint Language Trilogy: Semantics and Computations.* Technical Report CLS, Vancouver, Canada, 1988.

[46] D. Waltz. *Generating Semantic Descriptions from Drawings of Scenes with Shadows.* Technical Report AI271, MIT, Massachusetts, November 1972.

[47] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning.* Prentice Hall, Englewood Cliffs, New Jersey, 1984.