Towards Presentation Planning by Constructive Constraint Programming

Ulrich Scholz
European Media Lab GmbH
Schloß-Wolfsbrunnenweg 33
69118 Heidelberg, Germany
ulrich.scholz@eml-d.villa-bosch.de

Abstract

Constraint programming is a powerful and widely used technique to solve complex real-world problems by assembling and solving a corresponding constraint satisfaction problem. Standard constraint programming is restricted to problems with a fixed solution structure, which prevents its direct application to, *e. g.*, planning problems, for which the final solution structure is unknown until a solution is found. This paper introduces constructive constraint programming that overcomes this restriction by integrating the search for the solution structure into the search for a satisfying assignment. We outline the design of CoCoP, a generic implementation of a constructive constraint programming system, and describe leanCoP, a planning system that is the first step towards the implementation of CoCoP.

Motivation

The starting point of our work on constructive constraint programming is the problem of automated generation of multimedia presentations, which is one subproblem within the DynAMITE project[1]. Currently, we approach this problem with CkuCkuCk (André & Rist 1996), a planner specialized for this task. As the use of such a monolithic tool seems to be insufficient in the long run, we develop a system that is able to meet a wider range of requirements of presentation planning. Because of the complex nature of multimedia presentations this work leads us to an extension of constraint programming, a result that can be useful in other application areas.

The requirements of presentation planning are manifold and each of them can be approached with a specific technique. First of all, presentations are composed of timed subparts that depend on and interact with each other. A successful technique for such problems is constraint programming. Unfortunately, this approach disregards an essential aspect of a presentation: Its length and structure is unknown without having it finished. Consequently, the number of necessary variables and constraints to model it is unknown, too, and standard constraint programming is not applicable.

AI planning allows to search for the structure of the presentation, the property for which constraint programming fails. But again, this technique is not optimal because planning is only a small (yet inevitable) part of the overall complexity; most of presentation planning involves various classes of constraints, including geometric (screen layout), real valued (time), and integer valued constraints. Another requirement of presentation planning is the ability to call external functions. For example, if the planner decides to present an ASCII text via a speech engine, it often does not know the duration of the speech act. To measure its length, the planner has to employ an external tool to render the text into an audio format. Both features, support for diverse constraint solvers and external calls, are rare in current AI planners.

Two other attributes of a good presentation planner are extensibility and the capability to express subproblems in logic. As no two presentation settings are the same, the system might need a different set of constraint solvers for each new problem. Without being extensible, an existing system could suddenly become useless just because it cannot be adopted to a small change in the application domain. A further attribute is the seamless integration of logical subparts of a presentation problem that are best described in logic, *e. g.*, safety constraints and action preconditions. A perfect tool to generate presentations would handle them without the need to turn to clumsy encodings.

This paper documents our first thoughts on the first outline of a system that meets all the listed requirements. Hereby, we will concentrate on the basic design of the solver, rather than discussing specific problems of presentation planning, *e. g.*, how to model the problem or issues related to giving presentations. We begin by introducing constructive constraint programming, an extension of constraint programming towards planning. We then explain CoCoP, our anticipated constructive constraint programming system. Finally, we present leanPlan, a first step towards the realization of CoCoP.

Constructive Constraint Programming

Let us first repeat the definitions of a constraint satisfaction problem (CSP) and of constraint programming (CP): A CSP

[1] DynAMITE is concerned with intelligent multi-modal interaction with distributed networked devices within dynamical changeable ensembles, see http://www.dynamite-project.org.

consists of a set of variables and a set of constraints that restrict the assignment of values to these variables. A solution to a CSP is a satisfying assignment, *i. e.*, an assignment to all variables that satisfies all constraints. We do not regard variable domains as they can be subsumed under constraints. Constraint programming is the automated assembly of a CSP by a program, followed by a search for a satisfying assignment. It is a powerful technique to solve real-world problems because the flexibility of CSPs allows to model a large variety of problem classes and the application of informed search techniques allows to solve large CSPs.

One restriction of standard constraint programming is its limitation to problems with a fixed solution structure in respect to input size. In other words, the number of variables and the used constraints of the constructed CSPs have to depend solely on the size of the input. For example, applications of CP for cryptoarithmetics, such as for puzzles like SEND + MORE = MONEY, use exactly one variable per letter, independent of the frequency of a specific letter or its distribution in the puzzle. Consequently, standard constraint programming cannot be applied to, *e. g.*, planning problems where the solution size and structure is unknown before an actual solution is found.

We overcome this problem with constructive constraint programming (CoCP), an extension of constraint programming, which allows alternatives during the process of constructing the constraint satisfaction problem. With CoCP, the programming phase of CP turns in to a search in the space of CSPs. In each step of the search, the CoCP system chooses among a number of possible ways to further extend the current CSP. If the search phase terminates, the CoCP system has found a solution candidate; and if this candidate CSP has a satisfying assignment then the CoCP system has found a solution to the overall problem.

A constructive constraint programming system searches in two spaces, the space of CSPs and the space of satisfying assignments. As a result, having found a candidate CSP does not guarantee that there is a satisfying assignment to this CSP. Furthermore, if the construction of a candidate CSP succeeds but the search for a satisfying assignment of this CSP fails, then there can still be a solution to the overall problem with a different candidate CSP. Consequently, the CoCP system has to backtrack and try to find another candidate CSP. By interleaving both searches, the CoCP system can use the same techniques for the CSP construction that are used to speed the search for a satisfying assignment.

To give an example, consider the partial order planner UCPOP (Penberthy & Weld 1992), which can be seen as specialized constructive constraint programming system. This planner operates on plans, steps, and causal links. It starts with an empty plan, which only contains the initial state and the goal, and "systematically adds steps and constraints until it finds one that solves the planning problem." Adding a step includes the creation of causal links and ordering constrains, which encode the structure of the plan. This mechanism is a direct application of CoCP: A plan is basically a CSP; steps (the actions of UCPOP) are a set of variables (the action parameters) and constraints on these variables (equality constraints on parameters); and causal links are constraints on the equality of parameters of different actions. With the addition of steps to a plan, its variables and constraints are added to the CSP; the result is a new CSP. The algorithm terminates if all goals and subgoals are supported, *i. e.*, if all goal facts and all action preconditions are either added by an action or are present in the initial state. In other words, UCPOP intertwines the search for a satisfying assignment with the search for a candidate CSP.

CoCoP— Outline of a General Constructive Constraint Programming System

One aim of the DynAMITE project is to build a flexible tool for presentation planning. To this end we are designing the general constructive constraint programming tool CoCoP. CoCoP is flexible because its core search engine is independent of a specific problem class and its input language is first-order logic, extended by the ability to call external functions. This combination allows to attack a large variety of problems and avoids the problem of encoding procedural aspects and state transitions in logic while preserving the seamless integration of logical sub-problems.

The implementation language of CoCoP is Prolog. Many Prolog dialects provide various constraint packages as integral part of their language, so this choice enables the seamless integration of the search in the two spaces of CSPs and of satisfying assignments. Calls to external functions allow the direct use of these packages.

A front end specializes the core of CoCoP towards a specific problem class. Firstly, it provides a description how to (nonderteministically) build candidate CSPs from a problem specific input language. In contrast, a specialized CoCP system, *e. g.*, UCPOP is restricted to its dedicated problem class. Secondly, the front end provides a parser that translates a problem specific input language into the input language of CoCoP, allowing the core of CoCoP to stay problem independent. If the problem class changes, *e. g.*, if there is the necessity to use an additional constraint solver then CoCoP is adapted by changing the parser. This approach ensures the flexibility of the system and fosters rapid prototyping. Users of a specialized version of CoCoP interact with the system in their known problem specific input language; they do not have to learn about the internals of CoCoP. The next sections elaborate further on the core search engine and the input language of CoCoP.

CoCoP performs a forward search in the space of CSPs. At every choice point, it chooses one way to extend the current CSP until it finds a candidate one. If at one point the extension fails, it backtracks and tries a different option. Note that the search strategy in the target space of the problem is independent of the search in the space of CSPs: In principle, CoCoP can simulate all possible search techniques. For example, as a planning tool it can search the space of partial plans in the same way as it is done by UCPOP.

Deduction and the Connection Method

The core of CoCoP is leanCoP (Otten & Bibel 2003), a deduction system for first-order logic based on the connection method (Bibel 1987); its input language is predicate logic.

```
 1 prove(F) :- Time1 is cputime, make_matrix(F,M), prove(M,1),
 2             Time2 is cputime, Time is Time2-Time1, print(Time).
 3
 4 prove(Mat,PathLim) :-
 5      append(MatA,[Cla|MatB],Mat), \+ member(-_,Cla),
 6      append(MatA,MatB,Mat1),
 7      prove([!],[[-!|Cla]|Mat1],[],PathLim).
 8
 9 prove(Mat,PathLim) :-
10      nonground(Mat), PathLim1 is PathLim+1,
11      writeln(PathLim), prove(Mat,PathLim1).
12
13 prove([],_,_,_).
14
15 % CoCoP clause for executing theory literals
16 prove(['CALL'(Theorem_Literal)|Cla],Mat,Path,PathLim) :-
17     !,
18     call(Theorem_Literal),
19     prove(Cla,Mat,Path,PathLim).
20
21 prove([Lit|Cla],Mat,Path,PathLim) :-
22     (-NegLit=Lit;-Lit=NegLit) ->
23         ( member(NegL,Path), unify_with_occurs_check(NegL,NegLit)
24           ;
25           append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
26           append(ClaA,[NegL|ClaB],Cla2),
27           unify_with_occurs_check(NegL,NegLit),
28           append(ClaA,ClaB,Cla3),
29           ( Cla1==Cla2 ->
30                append(MatB,MatA,Mat1)
31                ;
32                length(Path,K), K<PathLim,
33                append(MatB,[Cla1|MatA],Mat1)
34           ),
35           prove(Cla3,Mat1,[Lit|Path],PathLim)
36         ),
37         prove(Cla,Mat,Path,PathLim).
```

Figure 1: Core of leanPlan, a first version of CoCoP. Besides the clause in lines 16 to 19, it is the unaltered code of the proof system leanCoP.

Of course, it is not the main task of constructive constraint programming system to decide the validity of logical formulas. Nevertheless, the language of logic is well suited as input for a generic solution system because it allows to state a large variety of problems.

The reasons for the choice leanCoP as base for CoCoP are manifold. To begin with, many problems have logical sub-parts, for example action preconditions. A deduction system allows to state these subproblems in a natural way without the need of clumsy encodings. Furthermore, leanCoP is easily extendible and the relationship between the steps of the proof system and their meaning in terms of the problem is accessible to the user (up to a certain degree). To support these claims, we first have to give a very short introduction into the connection method. Please refer to (Bibel 1987) for a thorough introduction.

The connection method is based on the concepts of clause, path, and connection. A connectionist prover first brings the input formula into disjunctive normal form, *i. e.*, into a disjunction of conjuncts.[2] Each such conjunct is called a clause; and a path is a selection of exactly one literal from each clause. A connection is a set of two literals with the same predicate symbol, where the one is an atom and the other is a negated atom. For example, consider the formula $\big(P \wedge (P \rightarrow Q)\big) \rightarrow Q$ in its disjunctive normal form $\neg P \vee P \wedge \neg Q \ \vee \ Q$. This formula has the two paths $\big\{\neg P, P, Q\big\}$ and $\big\{\neg P, \neg Q, Q\big\}$, which contain the connections $\big\{\neg P, P\big\}$ and $\big\{\neg Q, Q\big\}$, respectively.

The basic theorem of the connection method states that a

[2] Actually, this step is not essential for connectionist theorem proving. We mention it here because it simplifies the explanation of the connection method and it is required by leanCoP and therefore by CoCoP.

```
plan(P, S1, S2) :-
      S1==S2, P=[].
plan(P, S1, S2) :-
      apply(A, S1, S3), plan(P2, S3, S2), P=[|P2].
apply(A, S1, S2) :-
      action(S1, A), test_precond(A, S1), realize_effect(A, S1, S2).
plan_exists :-
      init(S1), plan(P, S1, S2), goal(S2), writeln(P).

init(S) :-
      % always succeeds, S is the initial state
goal(S) :-
      % succeeds if S is a state and the goal holds in S
action(S, A) :-
      % A is a nondeterministically chosen action, fails if all actions are tried
      % a heuristic function can use the state S to control the action selection
test_precond(A, S) :-
      % succeeds if A is an action, S is a state, and the precondition
      % of A holds in S
realize_effect(A, S1, S2) :-
      % succeeds if A is an action and S1 is a state.  Then S2 is the state
      % that results from realizing the effect of A in S1
```

Figure 2: A simple state-space forward-search planner. The clauses `init`, `goal`, `action`, `test_precond`, and `realize_effect` are problem dependent and are just shown as comments.

$$\forall p, s_1, s_2.\ \big(CALL(unify(s_1, s_2)) \wedge CALL(unify(p, [\,]))\big) \rightarrow plan(p, s_1, s_2) \tag{1}$$

$$\forall p, s_1, s_2.\ \big(\exists p_2, s_3, a.\ apply(a, s_1, s_3) \wedge plan(p_2, s_3, s_2) \wedge CALL(unify(p, [a|p_2]))\big) \rightarrow plan(p, s_1, s_2) \tag{2}$$

$$\forall a, s_1, s_2.\ \big(CALL(action(a)) \wedge CALL(test\_precond(a, s_1)), CALL(realize\_effect(a, s_1, s_2))\big) \rightarrow apply(a, s_1, s_2) \tag{3}$$

$$\exists s_1, s_2, p.\ CALL(init(s_1)) \wedge plan(p, s_1, s_2) \wedge CALL(goal(s_2)) \wedge CALL(writeln(p)) \tag{4}$$

$$\big((1) \wedge (2) \wedge (3)\big) \rightarrow (4) \tag{5}$$

Figure 3: Horn formulas that correspond to the planning algorithm of figure 2. The following literals are theory literals: Equality, *i. e.*, the Prolog built-ins "$=$" and "$==$" are implemented by explicit unification via the theory literal $unify$. The Prolog built-in $writeln$ is used directly. The theory literals $init$, $goal$, $action$, $test\_precond$ and $realize\_effect$ are implemented by the same Prolog clauses that are used for the planner in figure 2. A plan is modelled by a Prolog list of actions.

formula is valid if and only if every path through the formula contains at least one connection. Therefore, our example formula is valid. Basically, a connectionist theorem prover enumerates all paths through a formula in an efficient way and checks whether they contain a connection. This strategy allows to realize very small proof systems with reasonable performance that can be easily understood and extended. For example, the proof system leanCoP consists solely of the Prolog program in figure 1 (without the Prolog clause in lines 16 to 19).

Besides the mentioned advantages, the formulation of problems in logic has disadvantages, too. First of all, it is difficult to formulate procedural aspects of a problem. In the situation calculus, for example, a new situation is defined by stating its logical relationship to the previous one. Consequently, to check a property of a given situation, one has to chain back through all previous ones up to the initial situation. In "procedural" planning, each state is a term and a new state is created by copying the previous one and changing this copy according to the effect of an action.

Another disadvantage of a formalization in pure logic is that even common details like equality have to formalized by axioms. Consider the formula $P(a) \wedge a = b \rightarrow P(b)$. It is not valid on its own because "$=$" is an arbitrary binary predicate symbol without meaning; we have to add the axioms of equality to get the expected outcome. In addition, a theorem prover neither "calculates" a result, *e. g.*, a plan nor allows to call other programs; it simply decides the validity of a logic formula. The following section explains how the design of CoCoP avoids these disadvantages.

The Mechanics of CoCoP

The search mechanism of CoCoP is that of leanCoP; it is explained in the following (numbers in brackets are line numbers of the CoCoP code in figure 1). At the beginning,

CoCoP chooses a start clause (5-7), *i. e.*, a clause with solely positive literals. From left to right, it chooses each literal as the current one (21, 37). Either CoCoP has considered the current literal before (23), in which case it succeeds on this literal, or it searches top-down for a different clause with a connecting literal (25-27). In the latter case, it removes the literal from the new clause (28) and recurses on this clause (35). CoCoP succeeds on a clause if it success on all its literals (13); it terminates successfully if it succeeds on the initial clause (7). It regains completeness by using an iterative deepening search on the number of non-ground clauses (9-11, 32). Please refer to (Otten & Bibel 2003) for a more detailed explanation.

The search algorithm of CoCoP has a close resemblance with the execution of a Prolog program: The initial clause corresponds to the goal clause of Prolog; the evaluation order, left to right and top-down, mimics the Prolog flow of control. The correspondence is especially obvious if its input to the proof system is a conjunction of Horn clauses. Here, the negative literal of a clause corresponds to the head of a Prolog clause and the sequence of connections corresponds to the execution of a Prolog program.

A further property of CoCoP that is similar to Prolog is its ability to call Prolog functions. This ability allows CoCoP to achieve side effects, to manipulate terms, and to use theories. In particular, it allows to overcome the problems and deficiencies of a formalization in logic. External functions calls from CoCoP input are realized by introducing a new type of literals, called theory literals, which are actually the heads of Prolog clauses. They are distinguished from regular, *i. e.*, "logical" literals by being wrapped in the otherwise unused, unary literal `CALL`. If CoCoP encounters the literal `CALL` then it unwraps its argument and calls the corresponding function, see lines 16 to 19 of figure 1. The handling of regular literals remains unchanged and the execution of leanCoP and CoCoP on a formula without theory literals is the same.

Besides the similarities, CoCoP and Prolog differ in several respects: First, the input to CoCoP is an arbitrary formula in first-order logic, *i. e.*, it is not limited to a disjunction of Horn clauses. Horn clauses and theory literals allow to "program" CoCoP to perform any task that could be performed by a corresponding Prolog program. As the flow of control for an arbitrary formula is much less obvious than for a formula consisting of Horn clauses, this ability will probably used mainly to state logical subproblems of an overall problem. Other differences are the test whether a literal has been considered before and the use of iterative deepening.

As an example of the correspondence between the processing of CoCoP input and the execution of a Prolog program, consider the simple state-space forward-search planner in figure 2. If this Prolog program is started by calling the clause `plan_exists` then it searches the space of plan prefixes in a depth-first manner. If it finds a plan, *i. e.*, `goal(S)` succeeds in clause `plan_exists` then it outputs the plan and terminates. The last five Prolog clauses are problem dependent; we only give them as comments. Note that for the sake of simplicity, the planner does not use iterative deepening search and thus is not complete.

Figure 3 shows the same algorithm as input to CoCoP. The first four Horn formulas correspond to the first four Prolog clauses and the fifth is the combined input formula. The theory literals are implemented by the same Prolog clauses that are used for the planner in figure 2. CoCoP evaluates these formulas from left to right and top-down, *i. e.*, in the same order as Prolog. Furthermore, it uses iterative deepening search. If we assume that the Prolog program would use an iterative deepening search, too, then both tools would execute the same search steps in the space of plan prefixes and yield the same result.

First Application: Planning

A first application of CoCoP will be the planner leanPlan that simulates the planner FF (Hoffmann & Nebel 2001) on the input language PDDL 1.2 (McDermott *et al.* 1998). By now, we have implemented a version that uses hand coded input; we are about to complete the parser and started to implement the heuristic control. In particular, leanPlan will use a heuristic inspired by the FF heuristic and perform similar search steps. Of course, it will be much slower as the highly optimized FF but as this planner is powerful and PDDL is a standardized definition language for planning problems, this choice might allow leanPlan to solve planning problems other than of toy size.

leanPlan consists of the CoCoP core as given in figure 1 and a parser that reads PDDL input and translates it into CoCoP input. Furthermore, it defines some auxiliary functions, *i. e.*, theory literals that are the same for all planning problems. Basically, these are the functions `action`, `init`, `goal`, `test_precond`, and `realize_effect` as given in figure 2.

States are realized with the Flux system (Thielscher 2005), an implementation of the fluent calculus (Thielscher 1998). Flux is a programming method for the design of agents that reason logically about their actions and has been developed for robot control. In leanPlan it is used for AI planning the first time. The methods of Flux are called by leanPlan directly via theory literals; PDDL objects are represented as numbers; and PDDL types are sets of numbers. A typed PDDL variable is a Prolog variable that is constrained to the numbers of its type. Besides the test of properties of states and the creation of new states according to the effect of an action, Flux allows to reason about incomplete knowledge and sensor information.

Consider, for example, the effect of the action `remove` of the assembly domain[3] and its corresponding input to CoCoP; they are shown in figures 4 and 5, respectively. The remove action has the typed parameters `?part` and `?whole`, which are translated into the all-quantified variables `PART` and `WHOLE`. Both variables are restricted to the objects of their types by applying the constraint "`#::[2..13]`", which sets their domain to $\{2, \ldots, 13\}$. The inequality between two such variables is enforced by applying the constraint "`#=`".

[3]For the full assemply domain, please refer to the FF domain collection at `http://www.mpi-sb.mpg.de/~hoffmann/ff-domains.html`.

```
:effect (and (not (incorporated ?part ?whole))
          (available ?part)
          (when (and (not (exists (?p - assembly)
                             (and (part-of ?p ?whole)
                                  (not (incorporated ?p ?whole)))))
                     (not (exists (?tp - assembly)
                             (and (transient-part ?tp ?whole)
                                    (not (= ?tp ?part))
                                  (incorporated ?tp ?whole)))))
                (and (complete ?whole)
                     (available ?whole))))
```

Figure 4: Effect of the action `remove` of the assembly domain.

```
(all PART: (all WHOLE: (all Z:
    ('CALL'(PART#::[2..13]),'CALL'(WHOLE#::[2..13]),
     ('CALL'(effect(incorporated(PART,WHOLE),neg,Z)),
      'CALL'(effect(available(PART),pos,Z)),
      (
          (
              (ex P: ('CALL'(P#::[2..13]),
                   ('CALL'(precondition('part-of'(P,WHOLE),pos,Z)),
                    'CALL'(precondition(incorporated(P,WHOLE),neg,Z)))))
          ;
              (ex TP: ('CALL'(TP#::[2..13]),
                   ('CALL'(precondition('transient-part'(TP,WHOLE),pos,Z)),
                    'CALL'(TP#\=PART),
                    'CALL'(precondition(incorporated(TP,WHOLE),pos,Z)))))
          )
      ;
          ('CALL'(effect(complete(WHOLE),pos,Z)),
           'CALL'(effect(available(WHOLE),pos,Z)))
      )
     )
    ) => effect(remove(WHOLE,PART),Z))))
```

Figure 5: Part of the CoCoP input formula that corresponds to the effect of the action `remove` of the assembly domain, shown in figure 4. It is the actual output of the leanPlan parser, slightly edited for readability. The `effect` predicate collects the effects in the non-logical variable Z. After processing the whole effect formula, leanPlan creates a new state by applying all collected effects at once.

Further Directions and Open Questions

Our long-term objective in the development of constructive constraint programming and CoCoP is a version that fulfills the requirements of a presentation planner in the DynAMITE project (Elting & Hellenschmidt 2004). The completion of leanPlan is the first step on this way but the path in-between is not clear yet.

One likely next step is the extension of leanPlan towards PDDL 2.2 (Edelkamp & Hoffmann 2003), *i. e.*, towards a more expressive version of PDDL with numerical properties. Another possible direction is the exploration of additional search strategies and the simulation of other planning tools. Besides partial order planners, like UCPOP, we are interested in planners that search backwards from the goal, *e. g.*, GRT (Refanidis & Vlahavas 2002).

An open question is how to integrate heuristic control efficiently. For example, the FF heuristic is calculated in every step of FF and its efficient calculation is a crucial aspect in the performance of this planner. Heuristics are calculated from features of the problem and the current search state. As the CoCoP system searches in the space of CSPs and the aspects of the problem are encoded as features of these CSPa, *i. e.*, as variable and constraints, it is difficult to make these aspects available to the heuristic calculation in an efficient way.

Conclusions and Related Work

This work presents constructive constraint programming, an extension of standard constraint programming, which is applicable to problems whose solution requires both a search for a constraint satisfaction problem and for a satisfying assignment to this CSP. Starting from the requirements of presentation planning in the DynAMITE project we have developed the design of CoCoP, a generic constructive constraint programming system that combines extensibility and the ability to process first-order logic. As a first step towards CoCoP we are about to implement leanPlan, a planner that simulates the planning system FF.

The advantages and disadvantages of using CoCoP in comparison to a specialized solver are those of using a general tool in comparison to a solver that is optimized towards a specific problem. The latter has always a performance advantage while the former allows for rapid prototyping, is extensible, and decouples the development of different components. If it is necessary to solve many similar problems as fast as possible then the development of a specialized solver is the best choice; but in case one has to handle complex and changing problem classes and problems of medium hardness and size then a tool like CoCoP is advantageous. After one has settled for a design that is optimal to solve a specific problem class and speed remains to be the critical issue then it is always possible to get that extra bit of performance by an optimized reimplementation of the solver. For this reason we believe that CoCoP has the chance of being used not only in the definition and the construction of prototypes but also in applications and consumer products.

Constructive constraint programming is strongly related to the concept of a structural constraint satisfaction problem (SCSP) (Nareyek 1999). Similar to CoCP, a SCSP intertwines the construction of a candidate CSP with the search for a satisfying assignment. Valid construction steps are given by structural constraints, which are defined via a graph grammar. A specific constraint solver uses both sets of constraints, the structural and the regular ones, to search for a valid candidate CSP and its satisfying assignment.

Structural constraint satisfaction problems and constructive constraint programming differ in that the former leans more towards constraint satisfaction and the latter more towards programming. The structural constraints of SCSP are applied by a constraint solver that controls the construction process. For example, some SCSP versions use local search and allow the active removal of constraints from the current CSP as well as structural invalid CSPs. In contrast, the input to a constructive constraint programming system, together with that system, constitute a program that constructs a CSP. In principle, the user has total control over the construction process via the input formula. The difference becomes clear if we try to simulate different search strategies: In CoCP the user can program these search strategies while the control in SCSPs is with the solver.

Another related approach is the concept of a dynamic constraint satisfaction problem (DCSP) (Mittal & Falkenhainer 1990), an extension of standard CSPs. It allows variables and constraints to be active or inactive depending on the satisfaction of activation constraints. Inactive variables and constraints are disregarded in the search for a satisfying assignment.

Dynamic variables allow for a certain amount of flexibility. For example, one could model the solution to a planning problem by providing n variables, where the i-th variable determines the i-th action in a solution plan. In case an actual solution is of length $n-1$ or shorter then activation constraints deactivate all superfluous variables. The disadvantage of dynamic constraint satisfaction is that there is still a maximum solution size and the superfluous variables and constraints are ballast.

Various existing programs can be seen as specialized constructive constraint programming systems, including a large variety of planners. They are special purpose tools, which can, in principle, be simulated by CoCoP.

Acknowledgments

The authors would like to thank Stephen Creswell, Jens Otten, Joachim Schimpf, and Michael Thielscher for providing software and support. Many thanks to Christian Elting for valuable discussions.

This work was partly funded by the German federal ministry of education and research as part of the DynAMITE project (grant 01 IS C27 B) as well as by the Klaus Tschira foundation.

References

[André & Rist 1996] André, E., and Rist, T. 1996. Coping with temporal constraints in multimedia presentation planning. In *Proc. 13th National Conference on Artificial*

Intelligence, volume 1, 142–147. Portland, USA: AAAI Press/MIT Press.

[Bibel 1987] Bibel, W. 1987. *Automated Theorem Proving*. Vieweg. second, revised edition.

[Edelkamp & Hoffmann 2003] Edelkamp, S., and Hoffmann, J. 2003. Pddl2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Institute of Computer Science, University of Freiburg, Germany.

[Elting & Hellenschmidt 2004] Elting, C., and Hellenschmidt, M. 2004. Strategies for self-organization and multimodal output coordination in distributed device environments. In *Workshop on Artificial Intelligence in Mobile Systems*, 20–27.

[Hoffmann & Nebel 2001] Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

[McDermott *et al.* 1998] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C. A.; Ram, A.; Veloso, M.; Weld, D.; and Wikins, D. 1998. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

[Mittal & Falkenhainer 1990] Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proc. 8th National Conference on Artificial Intelligence*, 25–32. Boston, USA: AAAI Press/MIT Press.

[Nareyek 1999] Nareyek, A. 1999. Structural constraint satisfaction. In Faltings, B.; Freuder, E.; Friedrich, G.; and Felfernig, A., eds., *Configuration: Papers from the 1999 AAAI Workshop*. AAAI Press, Menlo Park, CA. 76–82.

[Otten & Bibel 2003] Otten, J., and Bibel, W. 2003. LeanCoP: Lean theorem proving. *Journal of Symbolic Computation* 36(1-2):139–161.

[Penberthy & Weld 1992] Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B.; Rich, C.; and Swartout, W., eds., *Proc. 3rd International Conference on Principles of Knowledge Representation and Reasoning*, 103–114. Cambridge, MA: Morgan Kaufmann.

[Refanidis & Vlahavas 2002] Refanidis, I., and Vlahavas, I. 2002. The MO-GRT system: Heuristic planning with multiple criteria. In *Proceedings of the Workshop at AIPS on Planning and Scheduling with Multiple Criteria*.

[Thielscher 1998] Thielscher, M. 1998. Introduction to the Fluent Calculus. *Electronic Transactions on Artificial Intelligence* 2(3–4):179–192.

[Thielscher 2005] Thielscher, M. 2005. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming* 5(1).