Structural Constraint Satisfaction

Alexander Nareyek
GMD — FIRST
German National Research Center for Information Technology
Research Institute for Computer Architecture and Software Technology
Rudower Chaussee 5
D - 12489 Berlin, Germany
alex@first.gmd.de

Abstract

Conventional constraint satisfaction problem (CSP) formulations are static. There is a given set of constraints and variables, and the structure of the constraint graph does not change. For a lot of search problems, though, it is not clear in advance what a solution's constraint graph will look like.

To overcome these deficiencies, we introduce the concept of structural constraints, which are restrictions on admissible constraint graphs. The construction of constraint graphs is based on the concept of graph grammars. This allows us to formulate and solve structural constraint satisfaction problems (SCSPs), handling combinatorial search problems without explicitly giving the solution's structure.

Introduction

Constraint programming has become a fast-growing and successful discipline addressing the solution of combinatorial search problems. However, expressiveness can still be substantially extended.

A CSP consists of a set of variables $x = \{x_1, \ldots, x_n\}$, where each variable is associated with a domain $d_1, ..., d_n$, and a set of constraints $c = \{c_1, ..., c_m\}$ over these variables. The domains can be symbols as well as numbers, continuous or discrete (e.g., "door", "13", "6.5"). Constraints are relations between variables (e.g. "x_a is a friend of x_b", "$x_a < x_b \times x_c$") that restrict the possible value assignments. As domains can be interpreted as unary constraints, they are not considered here. Constraint satisfaction is the search for a variable assignment that satisfies the given constraints.

A lot of problems cannot be stated this way. Often, it is not only a question of a satisfying variable assignment, but also of the graph structure itself.

Consider the problem of configuration. There may be endless possible numbers and kinds of components and relations. For example, the decision to use a hundred antennas for a cell phone network instead of fifty makes a great difference to the net's other components and structure (hand-over components, spatial distribution, possible frequencies, etc).

The approach of considering maximal structures, where substructures can be deactivated if they are superfluous (e.g., by a transformation with 0-1 variables (Nareyek & Geske 1996)), is suitable for problems with few variations only. Bigger problems call for mechanisms to adapt the underlying constraint graph itself.

The SCSP should not be confused with the dynamic constraint satisfaction problem (see (Verfaillie & Schiex 1994) for a brief survey). Dynamic constraint satisfaction tries to revise a variable assignment with given changes to the constraint graph and does not include structural changes as part of the search.

Composite CSPs (Sabin & Freuder 1996) aim at a similar extension of the conventional constraint satisfaction paradigm as SCSPs. A composite CSP expresses subgraph alternatives in a hierarchical way. This allows optimized search guidance, but requires manual preprocessing to build the hierarchy. The creation of the hierarchy it is often problematic, as completeness and appropriate structure are not always obvious.

Generation by a Graph Grammar

We must have a mechanism for describing the search space, such as the Cartesian product of all variables' domains in conventional CSPs. This can be achieved by the concept of algebraic graph grammars.

We continuously expand an empty start graph toward a possible constraint graph by the application of specific rules (*productions*). These productions have to ensure completeness. The decisions about which production rule to apply and where to apply it on the graph are similar to conventional CSPs' variable- and value-ordering decisions.

Graph Grammars

This section provides a fairly informal introduction to algebraic graph grammars (see (Ehrig, Pfender & Schneider 1973; Habel, Heckel & Taentzer 1996; Rozenberg 1997) for a detailed overview).

Algebraic graph grammars are a generalization of Chomsky grammars. A graph signature $GSig$ consists of the sorts of vertices V, edges E, and a label alphabet L. The operations of $GSig$ provide source and target vertices for every edge, $s, t : E \to V$, and map a label to every vertex and edge, $l_v : V \to L$ and $l_e : E \to L$.

A match m of graph g_1 to graph g_2 is a partial graph morphism that maps the vertices and edges of g_1 to g_2 such that the graphical structure and the labels are preserved. We use injective matches only.

A production P is a partial morphism between a left-hand side P_l and a right-hand side P_r, which provides information about which elements are preserved, deleted and created in the case of an application of the production. The identity of objects is marked by appended identifiers like `:1` (e.g., in the production in Figure 7). A production is applicable to a graph g, if there is a match of P_l to g.

A derivation g_2 from g_1 is the so-called *push-out* graph of an application of an applicable production P. The new graph g_2 is similar to g_1, but the elements of P_r that are not in P_l are added, and elements of P_l that are not in P_r are deleted.

The application of a production may require application conditions in addition. A negative application condition (NAC) is a morphism $P_l \rightarrow n$ that is satisfied if there is no morphism $n \rightarrow g$ such that $P_l \rightarrow n \rightarrow g = P_l \rightarrow g$. An NAC is represented by a convex dark area (e.g., in production P_{Less} in Figure 6). For multiple NACs, such as in production P_{Mouse_e} in Figure 8, the conjunction of the conditions must hold. A positive application condition (PAC) is a morphism $P_l \rightarrow p$ that is satisfied if there is a morphism $p \rightarrow g$ such that $P_l \rightarrow p \rightarrow g = P_l \rightarrow g$. A PAC is represented by a convex light area (e.g., in the left-hand side of Figure 11).

Graph Elements for SCSPs

A variable of the CSP is represented by a vertex with the label *Variable*. It is graphically depicted by a circular vertex. Constraints could be represented by edges, but there are constraint types that allow a variable number of variables to be included. These constraint types will be called *extensible constraints* (C_e) in contrast to *nonextensible constraints* (C_n), such that $C_e \cup C_n = C$.

As the number of variables incorporated for extensible constraints may vary throughout the search, there must be a simple mechanism to include/exclude variables for constraints. Hence, a constraint is also represented by a vertex, new edges to variables being added in order to incorporate them. A constraint vertex's label corresponds to the type of the constraint. A constraint is graphically depicted by a rectangular vertex.

An SCSP allows the existence of so-called object constraints. These constraints do not restrict the variables' values, but provide structural context information. For example, it must be known which two variables together form a two-dimensional coordinate object. Otherwise, a LINE constraint to check if the included coordinates are all on one line might consider any pairs of variables for the check. Object constraints act as a kind of structural broker between variables and conventional constraints. They are represented by a rectangular vertex with a dashed outline. Object constraints can be nonextensible as well as extensible, $O_e \cup O_n = O$, $C \cap O = \emptyset$. Constraints of C can be connected to variables and object constraints, whereas object constraints can be connected to constraints of C as well.

Edges are used to connect vertex elements. The role/position of a variable (or constraint when speaking about objects) within a constraint is often very important. Thus, an edge's label and direction can be used to express its role/position[1]. An edge's direction is indicated by an arrow and the label is displayed in the edge's middle (*NoLabel* if omitted).

Figure 1 shows an example graph with two nonextensible TRANSMISSION STREAM object constraints, an extensible SUM constraint that restricts the sum of ($\circ \rightarrow \square$)-connected variables to equal a ($\square \rightarrow \circ$)-connected variable, and a nonextensible LESS constraint that restricts the ($\circ \rightarrow \square$)-connected variable to be less than the ($\square \rightarrow \circ$)-connected variable. The extensible NON-OVERLAP constraint is redundant, as the other constraints already forbid the streams' overlap.

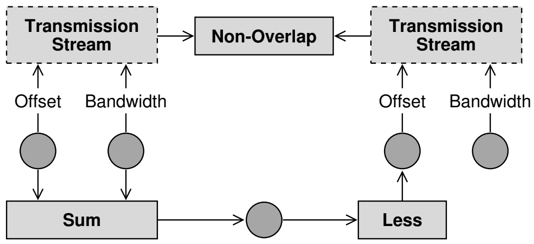

Figure 1: An Example Graph

Variables are always terminal symbols because the graph is merely expanded. This is also true for nonextensible constraints. However, extensible constraints may still be extended, and the final meaning of an extensible constraint is unclear until its expansion has ended. For the time being extensible, extensible constraints are represented by double-framed rectangular vertices that are nonterminal symbols. Edges cannot be extended and are always terminal.

Note that the distinction between nonterminal and terminal symbols is included as label information and is not realized by different vertex types. This allows generalized matches and the preservation of structural relations for productions that exchange nonterminal with terminal vertices.

A general vertex (terminal or nonterminal; variable, conventional constraint, or object constraint) is graphically depicted by a flat ellipse (see the right-hand side of Figure 11).

Structural Constraints

Structural constraints could be freely defined automata to test graphs for specific properties. However, to enhance computability, we use a stricter convention here.

[1] An edge's label could just as well be expressed by a special object constraint in between.

A conventional CSP's constraint correlates domain values. In contrast, a structural constraint correlates subgraphs. A conventional constraint's application point is defined by the problem formulation, whereas a structural constraint's application point is not clear in advance. Thus, a structural constraint needs a matching part that is equal to the left-hand side of a production rule (*docking part* S_d).

A conventional constraint is true as long as there is at least one tuple of possible variable assignments. There may be a few possible structures to be accepted by a structural constraint as well. Thus, structural constraints do not have only one right-hand side, like a production, but a set of alternatives (*testing part* S_t). These alternatives have a testing nature and are not used for pushouts like the production's right-hand side. Because of this, there may be application conditions not only for the docking part, but also for the testing part's alternatives.

Structural constraints are expressed by terminal symbols, but apply to any substitution with the corresponding nonterminal symbols as well. Figure 2 shows an example of the restriction that the frequencies of two base stations' antennas are different or that the stations' spatial coverages do not overlap.

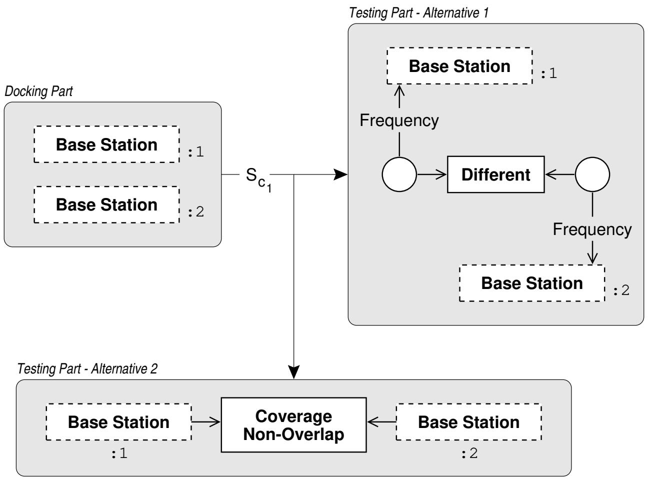

Figure 2: An Example of a Structural Constraint

A graph g is structurally consistent, if there exists a morphism $a \to g$, $a \in S_t$ for every structural constraint S and every possible match $S_d \to g$ such that $S_d \to a \to g = S_d \to g$.

(Heckel & Wagner 1995) introduce so-called *consistency conditions* that are equal to structural constraints, with a testing part consisting of one alternative only. These can be directly transformed into semantically equivalent application conditions of productions.

Structural Constraint Satisfaction Problems

A structural constraint satisfaction problem $SCSP = (\mathcal{CD}, \mathcal{S})$ consists of a tuple of sets of constraint descriptions $\mathcal{CD} = (\mathcal{C}_n, \mathcal{C}_e, \mathcal{O}_n, \mathcal{O}_e)$ and a set of structural constraints $\mathcal{S}$. The constraint descriptions of $\mathcal{C}_n$ and $\mathcal{O}_n$ are pairs (c, p_{base}) with a nonextensible conventional (or object) constraint c and its embedding graph p_{base}. The constraint descriptions of $\mathcal{C}_e$ and $\mathcal{O}_e$ are quadruple $(c, p_{base}, E, p_{max})$ with an extensible conventional (or object) constraint c, its minimal embedding graph p_{base}, a set of extension graphs E, and the constraint's maximal embedding graph p_{max}.

An embedding graph shows the constraint with all its directly connected neighbor vertices. If an extensible constraint has no maximal embedding, p_{max} is the empty graph. An extension graph shows the constraint connected to the vertices that can be added in one step. Figure 3 shows some components of an example SCSP.

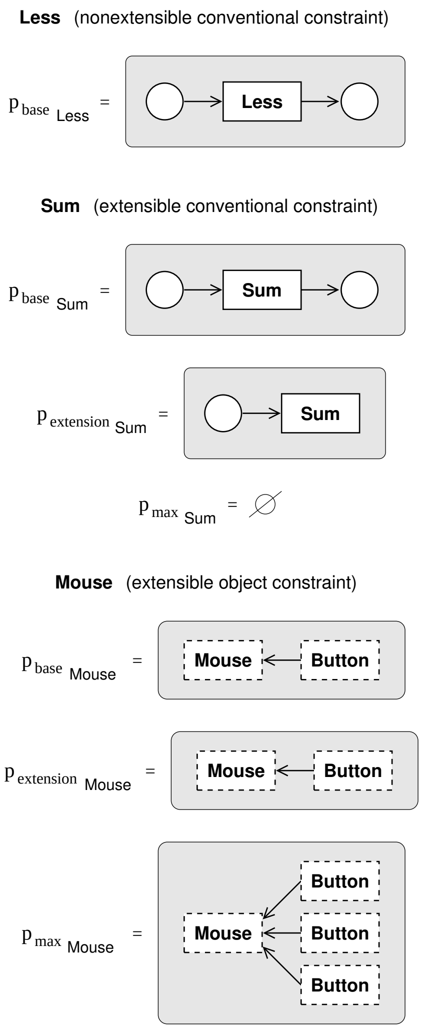

Figure 3: Some Components of an Example SCSP

There are some requirements that are induced by the construction of the search space in the following section:

- Nonextensible constraints are not allowed to appear in graphs of other constraints.
- Constraint-usage cycles are not allowed for the base embedding graphs of extensible constraints, e.g., that p_{base_A} includes a B constraint and p_{base_B} includes the A constraint.
- If the p_{max} graph of an extensible constraint is nonempty, no sequence of extensions that is applied to the constraint's p_{base} graph can produce a graph that includes the p_{max} graph without first producing the p_{max} graph.

Generating the Search Tree

This section describes how generic productions can be created using an SCSP formulation. These productions span the structural search tree. Figure 4 shows an example of a search tree.

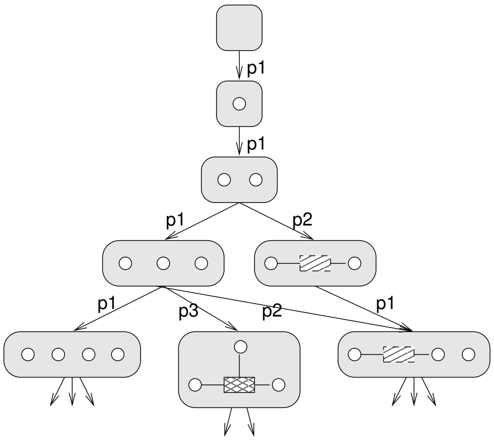

Figure 4: An Example of a Search Tree

We can, on the one hand, span a large general search tree and exclude invalid graphs by structural constraints; and on the other, we can create the productions in such a way that less invalid graphs are produced, thereby implicitly satisfying some structural constraints. This section provides automatically deducible productions that implicitly satisfy the structural constraints induced by the SCSP's embedding and extension graphs. Then, only the structural constraints of $\mathcal{S}$ must be taken into account during the search process.

Apart from improvements that are automatically deducible, numerous domain-specific improvements are possible for a specific SCSP. But according to conventional constraint satisfaction, we wish to have a *declarative* framework that does not rely on manually tailored improvements.

The following construction rules require that all terminal vertices be replaced by their nonterminal counterparts when speaking of embedding or extension graphs of extensible constraints.

- One production ensures that it is always possible to add further variables. The production is shown in Figure 5.

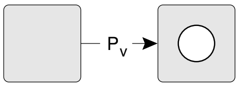

Figure 5: Addition of Variables

- There must be one addition production per nonextensible constraint, which is constructed in the following way:

  Construction $P_{nonextensible}$: The right-hand side of the addition production is equal to the constraint's embedding graph p_{base}. The left-hand side of the production contains the vertices of the right-hand side without the constraint itself, and an NAC that contains the right-hand side without the vertices that are connected to the constraint.

  Production P_{Less} in Figure 6 shows the production for the Less constraint.

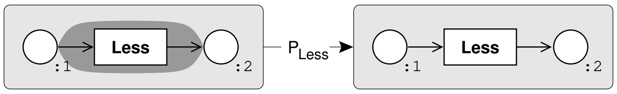

Figure 6: Addition of a Nonextensible Less Constraint

- Extensible constraints cannot be added in one step like the nonextensible constraints. Only the extensible constraint's p_{base} graph can be added at once, as this is the minimal structure. Thus, the addition production for an extensible constraint is similar to the one for nonextensible constraints:

  Construction $P_{extensible}$: The right-hand side of the addition production is equal to the constraint's embedding graph p_{base}. The left-hand side of the production contains the vertices of the right-hand side without the constraint itself.

  Production P_{Sum_b} in Figure 7 shows an example of the Sum constraint.

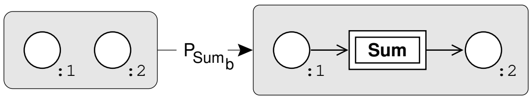

Figure 7: Addition of an Extensible Sum Constraint

- For the extensions of extensible constraints, there must be one production for every possible extension of an extensible constraint:

Construction $P_{extensible_e}$: The right-hand side of the production is an extension graph of E. The left-hand side is created by the vertices of the right-hand side, an NAC for each edge of the right-hand side, and, if p_{max} is not empty, a NAC consisting of the constraint's maximal embedding graph p_{max} such that the constraint is unified with the constraint of the right-hand side (NAC without the constraint itself). The same step to prevent a given maximum being exceeded must be taken for all other constraint vertices of the extension graph.

Production P_{Mouse_e} in Figure 8 shows an example of an extension of the MOUSE object constraint.

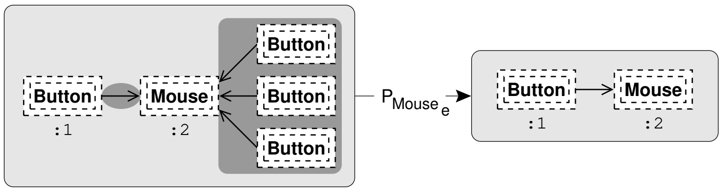

Figure 8: Extension of an Extensible MOUSE Constraint

- Every extensible constraint vertex must be turned into a terminal symbol at some time. There must be one production per extensible constraint, which is constructed in the following way:

  Construction $P_{extensible_t}$: The left-hand side consists of the nonterminal constraint vertex. The right-hand side shows the left-hand side's vertex as a terminal symbol.

  Production P_{Sum_t} in Figure 9 shows an example of the finalization of the SUM constraint.

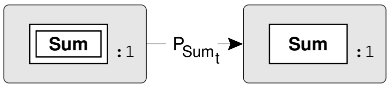

Figure 9: Finalization of an Extensible SUM Constraint

We can now span the search space with all graphs that can be produced by a sequence of production applications and that consist of terminal symbols only. The start graph is the empty graph.

Application of Structural Constraints

If the graph is changed by a production, the graph's consistency must be verified again. It would be very costly to test each time for all matches of possible structural constraints. Instead of this, *instances* of structural constraint *types* are memorized. These instances stay matched to a certain part of the constraint graph. Then, a re-verification of the graph must only be done with respect to the changes. A constraint instance is called

- *satisfied* if at least one of the test alternatives matches the graph and has no negative application condition
- *s-pending* if at least one of the test alternatives matches the graph and has negative application conditions that are satisfied
- *u-pending* if no test alternative matches the graph, but at least one of the test alternatives has no negative application condition that cannot be satisfied anymore
- *unsatisfied* if each alternative of the testing part has a negative application condition that cannot be satisfied anymore

See also Figure 10.

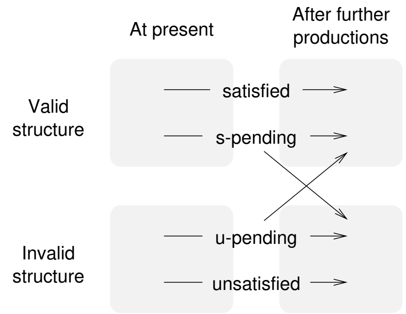

Figure 10: Properties of Structural Constraints

During search, the graph structure is continuously expanded. Whenever new elements are added by a production, there must be new structural constraint instances for all possible constraint matches that include the new elements. Since there might be negative application conditions in the docking parts of existing structural constraints, those with an unsatisfied negative application condition due to the new elements must be excluded.

After this addition and deletion of structural constraints, the remaining testing parts of relevant s- and u-pending structural constraints have to be checked. Alternatives with an unsatisfied negative application condition are excluded. If a structural constraint becomes unsatisfied and it has no NAC in its docking part, the search has reached a dead end. A solution is found if all constraints are s-pending or satisfied and there are terminal symbols only.

It is useful to integrate the conventional constraint satisfaction directly into the structural constraint satisfaction. As soon as a constraint of C is set as a terminal symbol and all connected object constraints (also via other object constraints in between) are set as terminal symbols as well, the constraint can be established for the conventional constraint satisfaction. The role/position of the constraint's variables is determined by the object constraints in between and the edges' labels and directions. The variables that do not already exist are created as well.

By the integration of conventional constraint satisfaction, consistency mechanisms like AC-4 (Mohr & Henderson 1986) can support the exclusion of inconsistent search paths as early as possible. The labeling of variables can start as soon as a solution to the structural problem is found. If the structural solution turns out to be inconsistent, the structural search has reached a dead end.

The branching factor of the search is very high, and pure backtracking does not work because of potentially infinite production sequences (breadth-first search is the only valid alternative for a complete search). In addition, structural constraints will rarely become unsatisfied, because it may often be possible to satisfy them by the inclusion of further elements. Consequently, search branches will be cut less often. Thus, we need powerful techniques and heuristics to support and guide the search. One of these techniques is the reduction of symmetry, such that similar structures are matched/transformed just once and not their equivalents again. On the other hand, there are numerous options for constructing heuristics, e.g., choosing productions such that u-pending constraints become s-pending.

Avoiding Redundancy

The addition of nonextensible constraints prevents redundant constraints by means of a corresponding NAC in $P_{nonextensible}$-productions. This avoidance of redundancy is not ensured for extensible constraints. Specific S-Constraints can overcome this problem. There must be an S-Constraint $S_{extensible}$ for each extensible constraint type, docking at each pair of potentially redundant constraints, and having all possible distinctive features as test alternatives.

Potentially redundant constraints are two constraints of the same type that are connected to the same elements according to the base graph p_{base}:

Construction $S_{extensible}$ — **docking part**: The docking part of the structural constraint is created by two p_{base} graphs, where the corresponding vertices (without the constraints themselves) are unified. To avoid multiple redundant structural constraint instances per potentially redundant constraint pair, all but the constraints themselves are a PAC.

Possible distinctive features are vertices that are connected to one constraint but not (in the same way) to the other:

Construction $S_{extensible}$ — **testing part**: There are two alternatives per unique edge (label; direction with respect to the constraint — toward it or away from it) that is included in the constraint's extension graphs. Each of the two alternatives consists of a graph with the two constraints of the docking part and an additional general vertex. Between each constraint and the general vertex is an edge corresponding to the unique edge. In the one alternative, the edge to the first constraint is an NAC; in the other alternative, the edge to the second constraint is an NAC.

Figure 11 shows an example of a Subset constraint that forces the set of (○→□)-connected variables to be a subset of the set of (□→○)-connected variables. The p_{base} graph of a Subset constraint includes two variables (○→□→○), and there are two possible extensions corresponding to the two possible variable connections.

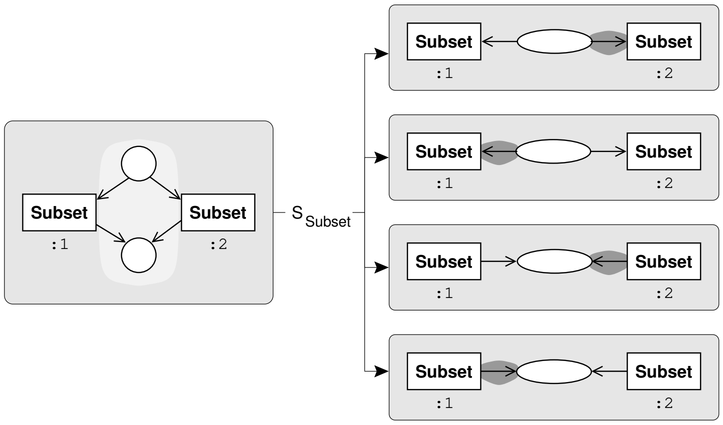

Figure 11: Avoidance of Redundant Subset Constraints

Conclusion

Combining conventional constraint satisfaction with structural requirements enables us to formulate and solve combinatorial search problems without explicitly giving the solution's structure. The SCSP approach follows the declarative constraint programming paradigm by stating only requirements for the solution without including information on solution generation.

The concept of structural constraint satisfaction contrasts with previous approaches that try to overcome the problem by considering maximal structures with deactivatable elements. The use of maximal structures is useful in the case of only slightly variable structures and a known maximum. A formulation by an SCSP does not have these restrictions.

The work done here based on the concept of algebraic graph grammars. The SCSP's description is used to generically produce productions and structural constraints, which are used to generate valid constraint graphs.

Further research topics include the introduction of more expressive structural constraints (such as the prevention of nonconnected graphs) and the combination of structural and conventional constraints. Another issue is the application of local search methods, which can usually conduct the search much faster and also provide approximate solutions before reaching a real solution. For conventional CSPs, one can argue in favor of refinement search as it can provide completeness and the verification of inconsistent problem descriptions. SCSPs, however, are undecidable, and we can neither achieve completeness nor verify inconsistency in finite time.

Acknowledgments

I wish to thank Gabriele Taentzer and Ulrich Geske for their useful comments and suggestions. This work is being supported by the German Research Foundation (DFG), NICOSIO, Conitec Datensysteme GmbH, and Cross Platform Research Germany (CPR).

References

Ehrig, H.; Pfender, M.; and Schneider, H. J. 1973. Graph Grammars: An Algebraic Approach. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT), 167–180.

Habel, A.; Heckel, R.; and Taentzer, G. 1996. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, Vol. 26, No. 3 & 4.

Heckel, R., and Wagner, A. 1995. Ensuring Consistency of Conditional Graph Rewriting - a Constructive Approach. In Proceedings of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA'95).

Mohr, R., and Henderson, T. C. 1986. Arc and Path Consistency Revisited. *Artificial Intelligence* 28(2): 225–233.

Nareyek, A., and Geske, U. 1996. Efficient Representation of Relations over Linear Constraints. In Proceedings of the Workshop on Constraint Programming Applications at the Second International Conference on Principles and Practice of Constraint Programming (CP96), 55–63.

Rozenberg, G. ed. 1997. *The Handbook of Graph Grammars. Volume I: Foundations.* Reading, World Scientific.

Sabin, D., and Freuder, E. C. 1996. Configuration as Composite Constraint Satisfaction. In Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop, 153–161.

Verfaillie, G., and Schiex, T. 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 307–312.