Bachelor's Thesis

Tiny Constraint Modelling Language

Marco Kuhlmann

4th June 2003

Universität des Saarlandes
Fachrichtung 6.2 – Informatik

Vorbemerkung und Erklärung

Diese Arbeit entstand als Abschlussbericht zu meinem im Sommersemester 2001 am Lehrstuhl von Prof. Gert Smolka an der Universität des Saarlandes in Saarbrücken durchgeführten Fortgeschrittenenpraktikum; Betreuer war Dr. Christian Schulte.

Hiermit erkläre ich, Marco Kuhlmann, an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 4. Juni 2003 Marco Kuhlmann

Abstract

Compared to the proprietary programming languages of systems like ILOG Solver, SICStus Prolog and Mozart Oz, constraint modelling languages offer a lot of benefits, such as usability and portability. This report presents the design and a prototype implementation of the "Tiny Constraint Modelling Language" (TCML). The data types and control structures of TCML have been designed to make the modelling of constraint problems intuitive and declarative; they include powerful iterators and arrays of dynamic size. The prototype implementation currently includes backends for Mozart Oz and ILOG solver.

Contents

1 Introduction

This report presents TCML, the "Tiny Constraint Modelling Language". TCML aims at providing a common interface language to constraint programming systems such as ILOG Solver, SICStus Prolog, or Mozart Oz. Why should one use a constraint modelling language, rather than the proprietary language that comes with a certain system, for constraint programming?

People just starting to learn constraint programming will have to spend quite some time to understand the new concepts (e. g., propagation, distribution, and search); they should not have to worry about design-specific issues. A modelling-language (like OPL, see [2]) can abstract away most of such things. Also, often constraint programming systems make use of languages that are relatively "exotic" to many users. But learning constraint programming *and* learning a new programming language at the same time is quite hard; it could be substantially simplified by the use of a modelling language which looks as familiar as possible to as many users as possible.

But constraint modelling languages also offer benefits to the experienced constraint programmer. Looking for the best combination of constraints for a given problem, one will often want to compare the performance of different development environments. Using a modelling language, the program in question only needs to be written once, and can then be tested on several systems. And also as an experienced programmer, one will welcome the possibility to focus on the essentials of constraint programming, and to be able to implement a constraint problem as declaratively as possible. That is what modelling languages are good for.

Finally, constraint modelling languages appear to be ideally suited for the application of automatic program analysis. If the language is kept relatively compact and simple, it should be easy to implement a static analyser for it - on the other hand, for many full-fledged programming languages, no analysis tools exist. Con-

straint programs written in the modelling language could then be type-checked even if their translation into the target language cannot. Even more interesting, one could aim at implementing a *constraint analyser* for the modelling language, which could for example analyse the propagation behaviour of constraints, suggesting better propagators, or detect obvious deficiencies.

As a modelling language, TCML was designed to make constraint programming as simple as possible: The syntax should quickly be familiar to anyone who has programming experience in any modern programming language. The data types and the control structures provided were chosen to make it straightforward to implement problem specifications in a declarative manner with minimal overhead. For example, TCML includes multi-dimensional dynamic arrays, which allow it to declare new variables "on the fly", avoiding tedious setups. Such arrays can be traversed using powerful iterators like `forall`, where in many proprietary programming language, an iteration over all elements of a multi-dimensional array would require manual conversion code.

This report is organised as follows: The next chapter introduces the central concepts and the overall design of the language. Chapter 3 elaborates on the prototype implementation of a small system translating TCML into the languages of two target constraint programming systems. The language design and the implementation are evaluated in chapter 4. Finally, chapter 5 gives a brief conclusion and outlook.

2 Language design

The starting point for TCML has been the study of a collection of common constraint problems in different languages and the identification of essential concepts of constraint programming languages that provide sufficient expressiveness. This section shall present the most important of these concepts.

Figure 2 shows the well-known "Send More Money" problem, as it can be stated in TCML. From the source code, one can already get a first glimpse of some of the features provided in the language. Like every other constraint language, TCML has to solve the following tasks: (a) represent problem-specific knowledge, (b) post constraints, (c) search the space of solutions that are obtained by constraint propagation. The successful achievement of task (a) depends on the availability of *data types* and their operations. For the task (b), *control structures* are important. Finally, the way one can search for solutions (task c) crucially depends on how the constraint problem as a whole is represented in the modelling language, the *metaphor* that is used for constraint satisfaction problems. In the following subsections, these topics are addressed successively.

2.1 Data types and operations

The main aim of TCML is to enable the user to concentrate on the essentials of constraint programming. Therefore, the number of data types is kept small but rich enough to provide a sufficient expressivity. The data types considered are integers (type `int`) and arrays (type `array(`t`)`), including nested arrays.

Integers come in two flavours: they can either have a fixed value, or be finite-domain integer variables. In the latter case, upon declaration, one has to specify a value domain for them (see line 2 of figure 2). Arrays can be nested; a two-dimensional

```
int[] money() {
   int s,e,n,d,m,o,r,y in {0..9};
   int[8] solution = [s,e,n,d,m,o,r,y];
   allDifferent(solution);
   s :!= 0;
   m :!= 0;
                1000*s + 100*e + 10*n + d
              + 1000*m + 100*o + 10*r + e
   := 10000*m + 1000*o + 100*n + 10*e + y;
   distribute(ff,solution);
   return solution;
}

void main() {
   search(all,money);
}
```

Figure 2.1: Send More Money

array of integers for example has the type `array(array(int))`. In the simple case, the dimensions of an array, that is, its depth and width at each level, are known upon initialisation; the one-dimensional array `solution` in figure 2 is a typical example. Complex problems require more elaborate arrays.

2.1.1 Dynamic arrays

The basic idea behind *dynamic* arrays is to free the programmer from the explicit and tedious setup of array sizes, as it often is needed to model constraint problems. In dynamic arrays, the array size is computed automatically as computation proceeds. A typical application of dynamic arrays is given in figure 2.2, which shows the famous Golomb Ruler problem [6] for a ruler of grade 5. In this example, the array d is used to store an upper triangular matrix of values $d_{ij} = k_j - k_i$ (lines 10-14). Then, the `allDifferent` constraint is applied to this matrix (line 15). What is the semantics of this constraint? Note that we cannot want it to enforce *all* the entries of the array d to be different, as that would include even those entries d_{ij} on the "empty half" of the matrix, which have not explicitly been assigned a value. It is here where dynamic arrays show their strengths.

```
int[][] golomb5() {
   int n = 5;
   int nn = n*n;
   int[n] k in {0..nn};
   int[][] d in {0..nn};
   k[0] := 0;
   foreach i in {0..n-2} {
      k[i+1] :> k[i];
   }
   foreach i in {0..n-2} {
      foreach j in {i+1..n-1} {
         d[i][j] := k[j]-k[i];
      }
   }
   allDifferent(d);
   distribute(naive,d);
   return d;
}
```

Figure 2.2: Golomb rulers of grade 5

The semantics of dynamic arrays is as follows:

- A new dynamic array is initialised by giving the number of its dimensions n and the element type t. For example, in figure 2.2 (line 5), the array d is initialised with $n = 2$ (the number of pairs of brackets after the `int` and $t =$ `int`). In this special case, the array is going to store finite-domain integer variables of a particular domain, which also has to be given at initialisation. In the following, this will be referred to as the "init-domain". After initialisation, a dynamic array contains nothing but the information about the type of entries that eventually will be included in it.

- Writing a value x to position i of the array will store x at that position and enforce x to be within the init-domain. Notice that if this was omitted, we would loose important information: Consider an array of finite-domain integer variables $v \in \{1, 2\}$. If we store a new variable $x > 1$ at some position of that array, and do not post the init-domain constraint, x would still have the domain $\{ n \mid n > 1 \}$, whereas it should be $x = 2$.

- Reading position i will store a fresh variable x of the init-domain at that position and return it, or return x, in case position i has been assigned this value previously.

With these semantics of dynamic arrays, `allDifferent` in the "Golomb" example can be made to affect only those positions i of the array d, which actually have been initialised in lines 10–14. In many other programming languages, one would have to precompute the number of variables that eventually will receive values, initialise a vector of the appropriate width, and map each matrix position (i, j) onto a position on that vector to post the constraints.

2.1.2 Linearisation

Compare the applications of the `allDifferent` constraint in figure 2 and figure 2.2. Whereas in the "Send More Money" problem, it is applied to a vector of finite-domain integer variables, in the "Golomb" problem its input is a two-dimensional array of such variables. This reveals the fact that the `allDifferent` constraint in TCML is polymorphic. However, full type polymorphism is only mimicked by introducing a special meta-type `lin(`t`)` for "linearisable types". For example, `allDifferent` has the input type `lin(array(int))`, which means that it accepts inputs that can be linearised to a vector of integers. Implicit linearisation is an important concept in a language like TCML, where one has both static types and arbitrarily nested arrays. It is not only employed for constraints like `allDifferent` and `distribute`, but also for the various iteration statements (see below).

2.2 Control structures

One of the major motivations and design goals for TCML is a familiar-looking syntax. This is most obviously realised in the selection of control structures, which makes TCML look a lot like Java: there are assignments, procedure calls, a `return` statement, and conditionals. To post constraints, TCML extends the Java syntax. Basic constraints are issued by prefixing a Boolean test operator (`=`, `!=`, `:<` etc.) with a colon (e. g., in line 5 of figure 2, the variable `s` is constrained to be different from zero). More complex constraints, like `allDifferent` and `distribute`, are represented by built-in procedures.

2.2.1 Iterators

Another extension of TCML are the different iteration statements. Iterators are very important in modelling constraint problems; a typical usage is the construction of the already discussed matrix d in figure 2.2.

TCML supports four different iteration constructs:

- `for` is like in Java or C.
- `foreach` x `in` X iterates over the elements of X (i. e., x is successively bound to each element of X).
- `foreach @`i `in` X iterates over the *indices* of X (i. e., for each element $x \in X$, i is bound to a natural number, representing the index of x in X. In arrays, the iteration index will be the same as the *real* index of x in the array, so we could also iterate over natural numbers between 0 and the size of the array. However, if X was a set rather than an array, then there would be no "natural" indexation.
- `foreach` x`@`i `in` X is the obvious combination of the latter two.

In all iterations, X has to be (bound to) a value that can be linearised into a vector of values of type t; x will then have type t.

2.3 Metaphors for constraint problems

Different constraint programming systems use different metaphors to represent constraint problems.

For example, ILOG Solver, being based on C++, has *objects* for the value environment to which constraints are posted, the distribution goal, and the search space. To add a constraint or determine a distribution or search strategy, one calls member functions of these objects. User-defined constraints are facilitated by functions that for example take a value environment as input, modify it by posting some additional elementary constraints, and return it as their value.

On the other hand, in Mozart Oz, constraint propagation and distribution are provided on the top-level as calls to procedures in the constraint library. Users can define

new constraints by procedures ("scripts") that produce as output the constrained variables. Search then is performed by applying these procedures to clones of the current system state (the "search space"), one for each choice point for a yet undetermined variable, which eventually builds up a search tree with search spaces that are either determined (i. e., all variables have been assigned a value) or failed as its leaves. For more on search in Mozart, see [4].

TCML employs the latter metaphor: As can be seen from figure 2 and figure 2.2, the constraint problem is encapsulated in a procedure returning an array of constrained variables. This procedure is then called from the `main` procedure by means of `search`. Although this requires first-class procedures, it seems to me the most natural because most declarative way of modelling constraint problems.

3 Implementation

To evaluate the expressiveness of TCML, a small prototype compiler for the language has been implemented, together with two backends that generate target language code for Mozart Oz and ILOG Solver. The architecture of this compiler can be seen in figure 3.1; it is a fairly standard separation into front-end (scanner and parser), middle-end (static and constraint analysis), and back-ends. The implementation will not be discussed in much detail here, as the source code comes with extensive documentation in the literate programming style. Rather, the focus will be on two components that have shown to be especially interesting.

3.1 The preprocessor

The main task of the preprocessor is the static analysis of a TCML program. Besides that, it also performs normalising transformations on the constraints. This is both needed for the constraint analyser, which relies on a constraint classification (see below), and is also useful in code generation, as different target systems often use different syntax for different types of constraints. Even more important, for some classes of constraints, there may exist more powerful propagators in the target system than for others.

The arithmetic constraints that we consider in TCML have the form e_1 op e_2, where e_1 and e_2 are arithmetic expressions, and op is a constraint operator. This can be rewritten as $e_1 - e_2$ op 0. The left hand side of that formula is an arithmetic expression itself, so it can be brought into the form $\sum_{i=1}^{n} d_i \prod_{j=1}^{m_i} x_{ij}^{p_{ij}}$, where the d_i are integers and the x_{ij} are (distinct) variables. The constant factors of this term can be summed up to yield one constant d, which can be transferred to the right hand side of the

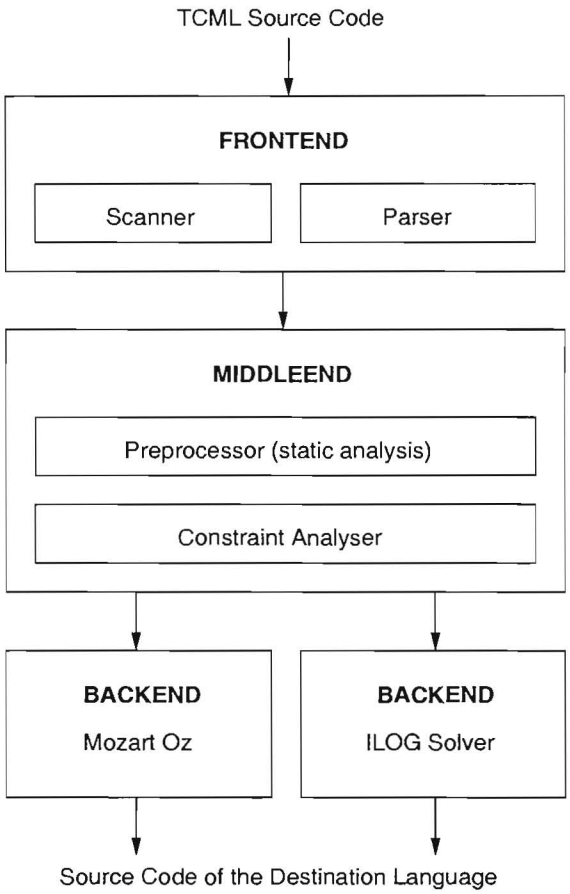

Figure 3.1: Architecture of the prototype implementation

equation. Finally, the scalars can be extracted, yielding the normal form

$$\sum_{i=1}^{n} d_i \prod_{j=1}^{m_i} x_{ij}^{p_{ij}} \operatorname{op} d.$$

Among the problems associated with constraint normalisation is the task to decide when two variables are distinct - the more identical variables one can identify in a constraint expression, the more simple it will become, and the more effective constraint propagation will be. However, the problem cannot be solved in the general case. Consider for example the variable expressions `foo[42]`, `foo[2*21]`, `foo[bar]` and boo. To find out that the first two are equal requires partial program execution; the preprocessor has to know that the expressions 42 and `2*21` denote the same value. When the variable `bar` is bound to the value 42, the first three variables become equal - but this is even harder to keep track of. Finally, boo might be a pointer to `foo[42]`, but how to tell? Here, one has to make a trade-off between the strength of propagation and the strength of the analysis techniques employed. A very interesting alternative would be to perform the normalisation at runtime, as then, all the variable equalities would be known.

After normalisation, several classes of constraints can be identified:

- linear constraints: $\sum_{i=1}^{n} d_i x_i \operatorname{op} d$
- linear constraints with unitary coefficients: $\sum_{i=1}^{n} \pm x_i \operatorname{op} d$
- simple products: $x_1 = x_2 \cdot x_3$
- squares: $x_1 = x_2^2$

These constraints, tagged with their class, form the input to the constraint analyser.

3.2 The constraint analyser

The basic idea behind the constraint analyser component is to extend static analysis to statements involving constraints. Like one can collect type information by annotating the atomic expressions of a program with types, and propagating this information up the parse tree, one should also be able to do something similar

with constraints: annotate the atomic constraints with information about their effect or behaviour, and project this information onto more complex, user-defined constraints.

As a first application of this idea, in the TCML project, *propagator analysis* was explored, which tries to answer the question if a certain constraint allows for different kinds of propagation. More specifically, [5] have shown that in some contexts, domain propagation without loss in propagation strength can be replaced by the computationally simpler bounds propagation. To obtain the relevant information from a given constraint program, methods of abstract interpretation on the constraint domains are employed.

For the prototype implementation of TCML, this analysis has been implemented as a module, that annotates the parse tree with the relevant information. During code generation, this information can be used to choose a source language command to post the constraint in question with just the right propagation strength. (Many constraint programming systems provide different implementations of a given constraint for different forms of propagation.)

The constraint analyser component has to face with a number of severe problems. For example, to apply the analysis methods that [5] suggest, it is necessary to know about the status of the variables that take part in a given constraint. But when it comes to constraints like `allDifferent`, which can be applied to dynamic arrays, it is not really clear how one can obtain this information – how does one find out what variables are currently stored in the array, and what domains they have? Another problem arises with the possibility to build user-defined constraints through procedures and loops. To guarantee a satisfactory analysis in these cases, one would have to apply methods of multi-variant analysis. Due to these problems, the constraint analyser has not yet been integrated into the system.

4 Evaluation

The implementation has been tested with a set of constraint problems, most of them from the Finite Domain Constraint Programming Tutorial found on the Mozart Oz web site [3]: The constraint problems were manually translated into TCML and then compiled to get code that could be evaluated in Mozart Oz and ILOG Solver. This caused no problems, and although the testing data was quite limited, TCML can be considered powerful enough to model a large class of constraint problems, and produce executable code for both target systems. As an example for a successful translation, see figure 4.1.

The effort that had to be put into the back-ends depended very much on the target language. For Mozart Oz, quite sophisticated support libraries were needed that implemented data structures (i. e., dynamic arrays) and provided auxiliary procedures (i. e., linearisation), while with these libraries available, the code generation was fairly straightforward. In contrast, for ILOG Solver, multi-dimensional arrays were already available as a data structure, but the code generation was much more involved – for example, a two-dimensional array in ILOG Solver is a one-dimensional array of one-dimensional-arrays; therefore, static information has to be used in much greater extent in that back-end than for (dynamically typed) Oz.

Mozart Oz

```
{TCML.allDifferent Osolution}
Os \=: 0
Om \=: 0
{FD.sumC
  [1 91 ~9000 ~90 ~900 10 1000 ~1] [Od Oe Om On Oo Or Os Oy]
  '=:' 0}
{TCML.distribute ff Osolution}
```

ILOG Solver

```
model.add(IloAllDiff(env,Isolution));
model.add(Is!=0);
model.add(Im!=0);
model.add(
  1*Id+91*Ie+(-9000)*Im+(-90)*In+(-900)*Io+10*Ir+1000*Is+(-1)*Iy
  == 0);
IloGoal goal = IloGenerate(env,Isolution,IlcChooseMinSizeInt);
```

Figure 4.1: Translation of the constraints from figure 2

5 Conclusion and outlook

This report presented a basic design for a constraint modelling language, and discussed several issues that had to be considered during its design and implementation. TCML enables its users to implement constraint problem specifications in a declarative fashion. It provides data types especially well-suited to constraint programming, like dynamic arrays, and powerful control structures to operate on them, like iterators with implicit linearisation. This reduces the programming overhead to a minimum. The prototype implementation contains an analyser component and backends for two widely-used constraint programming systems, and has been successfully applied to a number of example constraint problems.

Developing a language like TCML is a non-trivial task, and many interesting aspects could not be explored in the limited amount of time that was available for this project. However, it provided a much better understanding of how a constraint modelling language could look like, of the obstacles on the way towards it, and of possible solutions for these. This final section presents some ideas on the further development of TCML.

The next step certainly would be to more thoroughly test both the language design and the implementation. More and more complex constraint problems should be re-formulated in TCML, and the suitability of the language for modelling these problems should be evaluated. Although a modelling language cannot possibly be expected to provide all of the often very implementation-specific expressive power of a single constraint programming system, its coverage should be as broad as possible. Therefore, it should also be considered how to extend TCML. The most obvious of such extensions probably is to add support for more data types; for example, aiming at declarative programming, sets would be very useful. Also, more backends have to be written, supporting constraint programming systems like SICStus Prolog and Alice [1].

The single most interesting unit of the current implementation is the constraint analyser. When the TCML project was started, not much was known about propagator analysis and the programming techniques that were needed to implement this. The theoretical progress made since then gives raise to the hope that the constraint analyser eventually can be extended to form one of the core components of TCML, in which the advantages of a modelling language can fulfil their complete potential. The development of a working constraint analyser and its integration into the modelling language seems to need a complete research project on its own.

To live up to the aims of being a language for beginners in constraint programming, but even for everyday usage in development and benchmarking, a user-friendly interface to TCML is desirable. Such an interface could not only provide an accessible front-end for compilation, but also give feedback from the constraint analyser to the user, allowing him to develop and possibly enhance constraint programs interactively. One could even imagine an extension of the analyser to a tutoring system for constraint programming.

The prototype implementation of TCML developed during this project has been made publicly available as open source [7]. It is hoped that both the language design and the current implementation will proof flexible enough to facilitate many of the extensions suggested here, as it is most obvious that expressiveness and accessibility are the two major criteria that will determine the success of the language.

Acknowledgements

TCML is the result of my "Fortgeschrittenenpraktikum", carried out at the Programming Systems Lab of Saarland University in summer/autumn 2001 under the supervision of Christian Schulte. I would like to thank him for his guidance and patience, and the staff at the lab for such friendly and stimulating an atmosphere.

Bibliography

[1] The Alice Programming Language, `http://www.ps.uni-sb.de/alice/`.

[2] Pascal van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.

[3] The Mozart Programming System, `http://www.mozart-oz.org/`.

[4] Christian Schulte. *Programming Constraint Services.* Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000. To appear in Lecture Notes in Artificial Intelligence, Springer-Verlag.

[5] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space. In Harald Søndergaard, editor, *Third International Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, September 2001. ACM Press.

[6] Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Modelling the Golomb Ruler Problem. In *Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99)*, 1999.

[7] TCML, `http://www.ps.uni-sb.de/~kuhlmann/projects/tcml.html`.