Compiling and Executing Declarative Modeling Languages in Gecode

Raffele Cipriano, Agostino Dovier, and Jacopo Mauro

Univ. di Udine, Dip. di Matematica e Informatica.
(cipriano|dovier)@dimi.uniud.it

Abstract. We developed a compiler from SICStus Prolog CLP(FD) to Gecode as well as a compiler from MiniZinc to Gecode. We compared the running times of the executions of (standard) codes directly written in the three languages and of the compiled codes for a series of classical problems. Performances of the compiled codes in Gecode improve those in the original languages and are comparable with running time of native Gecode code. This is a first step towards the definition of a unified declarative modeling language for combinatorial problems that will allow the user to define problem instances and solving meta-algorithms using high-level declarative language, and then to automatically translate the specification into a low-level encoding, that allows the solving algorithms to run efficiently.

1 Introduction

Combinatorial problems like planning, scheduling, timetabling and, in general, resource management problems, are daily handled by industries, societies and research companies. Although, in principle, problems in this family can be mathematically modeled and then solved by computers, the intrinsic NP-hardness of most of them prevents us from expecting from a tool an optimum solution in acceptable time. Therefore, we must focus on developing techniques for finding "good" solutions (as much as possible close to the optimal one) in acceptable time. In the past decades a lot of techniques and solvers have been developed to cope with this kind problems, like branch and bound/branch and cut/branch and price algorithms, constraint programming techniques, local search, heuristics, an so on. Moreover, several modeling languages have been proposed to easily model problems and instances, and to easily interact with the solvers. In fact, it would be desirable to work with a user-friendly modeling language that allows the user to define the problem in an easy and flexible way interfaced with an efficient solver able to explore in a clever way the space of the solutions.

In this work we have focused our attention on the Gecode solver, a recent C++ constraint solving platform with excellent performances [11] and on declarative modeling languages, like constraint logic programming [13] and MiniZinc [15]. Encoding constraint satisfaction/optimization problems using the C++ framework Gecode is not very user friendly as using high/medium-level declarative modeling languages. We present a compiler from SICStus Prolog CLP(FD) [1] to Gecode as well as a compiler from MiniZinc to Gecode. We have chosen a set of (well-known) benchmarks, and we

have compared their running times in the original paradigms (SICStus, MiniZinc, and Gecode) and the running time in Gecode of their translation. We also compared our results with the translation of MinZinc into Gecode offered by MiniZinc developers.

The results are rather encouraging. Native code executions are typically faster in Gecode than in SICStus and MiniZinc (in the case of SICStus a consistent comparison is reported in [11]). However, in all cases, compilation (and then execution) in Gecode improves the performance of the native execution, and, moreover, these times are comparable with running time of native Gecode code.

This would allow the user to model problems at high level keeping all the well-known advantages of this programming style, without loosing efficiency w.r.t. C++ encoding. Moreover, our encoding of MiniZinc in Gecode outperforms the similar translation (via FlatZinc) presented in [10].

The present work is part of a general project of developing a programming tool for combinatorial problems. This tool will be made of three main part: the modeling one, the translating one, and the solving one. In the modeling part the user will define in a high-level style the problem and the instance he wants to solve and the algorithm to use (constraint programming search, eventually interleaved with local search, integer linear programming, heuristics or meta-heuristics phases). In the translating phase the model and the meta-algorithm defined by the user will be automatically compiled into the solver languages, like Gecode or other ones. During the third phase, the overall compiled program will be run and the various solvers will interact in the way specified by the user in the first phase, to find the solution for the instance of the problem modeled. We are planning to test the tool on different families of problems including those we have already cope with: a hospital rostering (timetable) problem [2], the protein structure prediction problem [5, 3], and the planning problem [9].

The paper is organized as follows. In Section 2 we describe the languages used to model and solve our problems, starting from the high level ones (SICStus Prolog and MiniZinc, respectively in subsections 2.1 and 2.2) and ending with the low-level encoding in Gecode (subsection 2.3). In section 3 we explain how we perform the translation from the high-level languages to the low-level one: this process passes through an intermediate language (named CNT) that we describe in subsection 3.1; the translation from SICStus Prolog and Minizinc code into CNT are described in subsections 3.2 and 3.3; the translation from CNT to the C++ final encoding in Gecode is described in 3.4. In section 4 we present the problems and the instances we used to test the solving performances of the different encodings (native SICStus Prolog, Minizinc and Gecode code and the various compiled code). Section 5 summarizes the results and explains the future works we intend to achieve. In Appendix we report the Prolog codes of the test problems used.

2 The languages used

We briefly introduce the high-level declarative language CLP(FD), the medium-level declarative modelling language MiniZinc, and the constraint solving platform Gecode. We use the N-Queens problem to briefly explain the different programming paradigms.

The N-Queens problem *It is the problem of putting N chess queens on an NxN chessboard such that none of them is able to capture any other using the standard chess queen's moves.* A standard way to model this problem is to consider that on the chessboard there will be one queen for each chessboard column: so we need N variables $X_1 \dots X_N$ (one for each chessboard column), with domain $X_i = \{1, 2, \dots, N\}$, with $X_i = k$ meaning that *the queen of column i will be placed on row k*. With this model, vertical attacks are implicitly encoded; to avoid horizontal attacks we must impose that $X_i \neq X_j \forall i \neq j$; to avoid diagonal attacks, we must impose that $\forall_i \forall_{j>i}, X_i \neq X_j + (j-i) \wedge X_j \neq X_i + (j-i)$.

2.1 CLP(FD)

$CLP(\mathcal{D})$ is a declarative programming paradigm, parametric on the constraint domain $\mathcal{D}$, first presented in 1986 [12] (see e.g. [13] for a popular review). Combinatorial problems are usually encoded using constraints over *finite domains* (namely, $\mathcal{D} = FD$), currently supported by all CLP systems based on Prolog (for example BProlog [17], GNUProlog [7], SICStus Prolog [1], ECLiPSe [16], just to name a few). We focused on the library `clpfd` of SICStus Prolog [1], but what we have done can be repeated for other systems embedding constraints on finite domains. We focus on the classical *constraint+generate* programming style, where a constraint definition phase anticipates a labeling stage. The SICStus primitives for defining an array of variables of length N (named Queens) and to set the domain of these variables in the integer range $1 \dots N$ are:

```
(1)    length(Queens, N),
(2)    domain(Queens,1,N),
```

The avoiding-horizontal-attacks constrain can be easily posted using the built-in constrain that post inequalities for each pair of variables:

```
(3)    all_distinct(Queens),
```

Then we impose that each queen (extracted by the recursive predicate *diagonal*) satisfies the avoiding-diagonal-attacks constrain (post with predicate *safe*):

```
(4)    diagonal([]).
(5)    diagonal([Q|Queens]) :-
(6)       safe(Q, 1, Queens),
(7)       diagonal(Queens).
(8)    safe( _,_,[]).
(9)    safe(X,D,[Q|Queens]) :-
(10)      X + D #\= Q,
(11)      Q + D #\= X,
(12)      D1 is D + 1,
(13)      safe(X,D1,Queens).
```

The D variable represent the difference $(j - i)$ of the model explained above; the `#\=` symbol is the standard SICStus Prolog way to post an inequality constrain over

variables (it is sufficient to add the # character before any standard relational symbol, such as =, \=, <, =< and so on).

2.2 MiniZinc and FlatZinc

MiniZinc is a medium-level modeling language developed by the NICTA research group [15]. It allows to express most CP problems easily, supporting sets, arrays, user defined predicates, some automatic coercions and so on. But it is also low-level enough to be easily mapped onto existing solvers. It is a subset of the language Zinc.

To encode the N-Queens model in MiniZinc we must first declare an array (here named q) of N variables with domain in the integer range $1 \ldots N$, using:

```
(14)  array [1..n] of var 1..n: q;
```

Then we must post the constraints on all the couple of variables, using the code below:

```
(15)  constraint
(16)     forall (i in 1..n, j in i+1..n) (
(17)        noattack(i, j, q[i], q[j])
(18)     );
```

The constraints are posted by the predicate `noattack` (line 21 deals with horizontal attacks, lines 22-23 deal with diagonal ones):

```
(19)  predicate
(20)     noattack(int: i, int: j, var int: qi, var int: qj) =
(21)        qi != qj ∧
(22)        qi + i != qj + j ∧
(23)        qi - i != qj - j;
```

FlatZinc is a low-level solver-input language, and it is mostly a subset of MiniZinc. The NICTA research group provides a compiler from MiniZinc to FlatZinc that support all solver-supported global constraints. This way, a solver writer can support MiniZinc with the minimum effort of providing a simple FlatZinc frontend to the solver and combining it with the existing MiniZinc-to-FlatZinc translator. The NICTA team also provide its own solver that can read and execute a FlatZinc model.

2.3 Gecode

Gecode is an open, free, portable, accessible, and efficient environment for developing constraint-based systems and applications (see [11] for details). It is implemented in C++ and offers competitive performances w.r.t. both runtime and memory usage. It implements a lot of data structures, constraints definitions, and search strategies, allowing also the user to define his own ones. Its modeling language style is C++ like, and, thus, programmer should take care of several low-level details. We report an extract of the N-Queens problem encoded in Gecode, regarding the horizontal and diagonal constraints.

```
(24)   for (int i = 0; i<n; i++){
(25)      for (int j = i+1; j<n; j++) {
(26)         post(this, q[i] != q[j]);
(27)         post(this, q[i]+i != q[j]+j);
(28)         post(this, q[i]-i != q[j]-j); } }
```

To simplify the using of the Gecode solver, its developers provide a FlatZinc frontend, i.e. an executable program that reads a FlatZinc model, solve it using the Gecode libraries and prints the solution (if any) on the standard output.

3 Translation

The translation from SICStus and MiniZinc programs to Gecode is carried on in two stages: first we translate the high-level code into an intermediate language (called CNT) that lists explicitly all the constraints. Then we generate C++ code from the CNT file, using static analysis to improve the second part of the compilation. After a brief introduction to the language CNT, we show the main lines of this two-steps translation.

3.1 CNT

The intermediate language CNT is used for listing the constraints, specifying some searching parameters and which variables should be printed. The CNT grammar is the one described in Table 1 (the grammar is also defined in the file `parser.y` [4]).

In the CNT language the variables are not typed, in sense that every variable is assumed to be an integer variable. Boolean variables are special variables that can assume only the values 0 and 1. This CNT language was realized to ease the listing of the constraints defined by a prolog program. For that reason we didn't add the possibility to declare variable arrays.

The semantics of the CNT language is intuitive. We only report the semantics of the non-terminal <sentence>:

rule 3: "domain $[x_1, \ldots, x_n]$, n_1, n_2" sets the domain $[n_1 \ldots n_2]$ to the FD variables $x_1, \ldots, x_n$.
rule 4: "in x, n_1, n_2" sets the domain $[n_1 \ldots n_2]$ to the unique FD variable x
rule 5: "sum $[x_1, \ldots, x_n]$, *op*, x" posts the constraint: $\sum_{i=1}^{n} x_i \sim_{op} x$
rule 6: "scalar_product $[n_1, \ldots, n_k]$, $[x_1, \ldots, x_k]$, *op*, x" posts the constraint: $\sum_{i=1}^{k} n_i * x_i \sim_{op} x$
rule 7: "all_different $[x_1, \ldots, x_n]$" adds the all_different global constraint between the variables $x_1, \ldots, x_n$ (viewed in binary form: $\forall i \forall j \; 1 \leq i < j \leq n . x_i \neq x_j$)
rule 8: "element $[x_1, \ldots, x_n]$, y ,z" sets the constraint $x_y \sim_{eq} z$
rule 9: bool expression b sets the (reified) constraint $b \sim_{eq} true$
rule 10: "write string_or_variable" print on the standard output the strings and variables passed as arguments
rule 11: "branch $[x_1, \ldots, x_n]$" adds $\{x_1, \ldots, x_n\}$ to the set of the variables to search

<S>	:= <sentence>	(rule 1)
	| <S> <sentence>	(rule 2)
<sentence>	:= domain <array of el> , NUMBER , NUMBER	(rule 3)
	| in <el> , NUMBER , NUMBER	(rule 4)
	| sum <array of el> , <relOp> , <el>	(rule 5)
	| scalar_product <array of int>, <array of el> , <relOp> , <el>	(rule 6)
	| all_different <array of el>	(rule 7)
	| element <array of el> , <el> ,<el>	(rule 8)
	| <boolE>	(rule 9)
	| write <string or variable list>	(rule 10)
	| branch <array of el>	(rule 11)
<boolE>	:= <el>	(rule 12)
	| not <boolE>	(rule 13)
	| (<boolE> <boolOp> <boolE>)	(rule 14)
	| (<arithE> <relOp> <arithE>)	(rule 15)
<arithE>	:= <el>	(rule 16)
	| abs(<arithE>)	(rule 17)
	| max(<arithE> , <arithE>)	(rule 18)
	| min(<arithE> , <arithE>)	(rule 19)
	| (<arithE> <arithOp> <arithE>)	(rule 20)
<boolOp>	:= and | or | xor | eqv | imp | reverse_imp	(rules 21-26)
<relOp>	:= > | < | <= | >= | == | !=	(rules 27-32)
<arithOp>	:= + | - | * | / | mod	(rule 33-37)
<el>	:= NUMBER | variable	(rules 38-39)
<variable>	:= _NUMBER	(rule 40)
<array of el>	:= [<list of el>]	(rule 41)
<list of el>	:= <el>	(rule 42)
	| <el>, <list of el>	(rule 43)
<array of int>	:= [<list of int>]	(rule 44)
<list of int>	:= NUMBER	(rule 45)
	| NUMBER , list of int	(rule 46)
<string or variable list>	:= <string or variable>	(rule 47)
	| <string or variable> <string or variable list>	(rule 48)
<string or variable>	:= STRING | variable	(rules 49-50)

Table 1. CNT grammar specification.

An example of CNT code is the following:

```
(29)  domain [_1, _2, _3, _4], 1, 4;
(30)  all_different [_1, _2,_3, _4];
(31)  ((_1 + 1) != _2);
(32)  ((_2 + 1) != _1);
(33)  ((_1 + 2) != _3);
(34)  ((_3 + 2) != _1);
(35)  ((_1 + 3) != _4);
(36)  ((_4 + 3) != _1);
(37)  ((_2 + 1) != _3);
(38)  ((_3 + 1) != _2);
(39)  ((_2 + 2) != _4);
(40)  ((_4 + 2) != _2);
(41)  ((_3 + 1) != _4);
(42)  ((_4 + 1) != _3);
```

The CNT language can be seen as a subset of FlatZinc and in the future we hope to bypass the use of the CNT language and produce for every program a FlatZinc file.

3.2 CLP(FD) into CNT

For translating SICStus to CNT, we automatically create a new SICStus program where constraints definition is replaced by a printing stage. For instance consider the following code in which we use the constraints "domain" and "all_different":

```
(43)  test(X,N) :-
(44)     length(X,N),
(45)     domain(X,1,N),
(46)     all_different(X).
```

To obtain the modified program we change all the predicates that add the constraints into the predicate "format" for printing informations. The previous code can therefore be converted into the following program:

```
(47)  test(X,N) :-
(48)     length(X,N),
(49)     format("domain ~q, ~q, ~q;\n", [X,1,N]),
(50)     format("all_different ~q;\n", [X]).
```

Some problems arise when there is a unification. In fact in some programs the logic variables are known to be FD-variables only at runtime and therefore every time in the program there is a unification we have to add some equality constraints. We developed some particular cases to cope with this and other minor technical problems.

The execution of the modified SICStus code instead of adding constraints simply prints all the constraints in the CNT form. Thus the execution of the modified program generates the CNT (flat) code. For instance, the execution of the modified version of the SICStus N-Queens code (1)–(13), with $N = 4$ generates the CNT code (29)–(42).

3.3 MiniZinc (FlatZinc) into CNT

We took advantage of the existing compiler from MiniZinc to FlatZinc [15] and thus focus on the translation from FlatZinc to CNT. As we said before the CNT language can be seen as a subset of FlatZinc, and thus, the subset of the FlatZinc programs that encode an integer CSP problem can be translated straightforwardly. Consider, for example, the following constraints in FlatZinc:

```
(51)  array[0 .. 2] of var 0 .. 2: v;
(52)  constraint int_eq(v[0], 0);
(53)  constraint all_different([v[0], v[1], v[2]);
```

These constraint can be defined in CNT in the following way:

```
(54)  domain [_0, _1, _2], 0, 2;
(55)  (_0 == 0);
(56)  all_different [_0, _1, _2];
```

3.4 CNT into Gecode

We have developed a compiler from CNT to C++/Gecode. Before the compilation we execute some simple simplifications to the CNT code. First of all we precompute numerical expressions and then we use the equality constraints to reduce the number of variables (using this expedient we are able to get rid of the useless constraints added in the prolog to CNT translation).

Moreover, using static analysis, the compiler groups the constraints that can be defined within a `for` cycle to reduce the final length of the C++ program. In some cases this lead to a dramatic reduction of time needed by Gecode for the compilation of the .cc file with its libraries. For instance, the Gecode file obtained by the instance 100-Queens with this optimization has a size of 53 KB and is compiled with the Gecode libraries in 5.8s, while the code obtained by "flat" CNT has a size of 1.5MB and requires 13 hours and 40 minutes for the compilation.[1]

Precisely, we have defined a syntactic notion of *matching* between constraints. Intuitively, two constraints match if they are based on the same predicate symbol and differ only in the index of some variables or in the other numbers involved. Two constraints that match can be defined in a "for" cycle (the definition of the matching function is correct but not complete in the sense that some constraints that could be defined in the same "for" cycle don't match). After partitioning the set of the constraints using the matching function, we analyzed each subset of the partition trying to group the constraints to declare each constraint with the minimum amount of "for" cycles. We realized that, due to the complexity of the problem, this task can be unfeasible. For this reason we realized a procedure that use the order of the definitions of the constraint to group them in a good solution. This idea tries to exploit the fact that a program declares the constraints following some principles.

[1] To be honest, we have been negatively impressed by the time required for compiling a Gecode specification on a Linux 64bit machine with 2.2 GHz. We think that Gecode developers should work on this problem.

To see how constraints can be defined in "for" cycles consider the following code in which for the first 9 elements in a list of length 10 we add the constraints that every element should be minor and the predecessor of the following element in the list.

```
(57)  example :- length(X, 10),
(58)     domain(X, 1,10),
(59)     constraint(X),
(60)     labeling([ff], X),
(61)     write(X).
(62)
(63)  constraint([_]):- !.
(64)  constraint([X,Y|Xs]):-
(65)     X #< Y,
(66)     X + 1 #= Y,
(67)     constraint([Y|Xs]).
```

If you launch the tool SICStus to CNT you will obtain the following CNT code:

```
(68)  domain [_1, _2, _3, _4, _5, _6, _7, _8, _9, _10], 1, 10;
(69)  (_1 < _2);
(70)  ((_1 + 1) == _2);
(71)  (_2 < _3);
(72)  ((_2 + 1) == _3);
(73)  (_3 < _4);
(74)  ((_3 + 1) == _4);
(75)  (_4 < _5);
(76)  ((_4 + 1) == _5);
(77)  (_5 < _6);
(78)  ((_5 + 1) == _6);
(79)  (_6 < _7);
(80)  ((_6 + 1) == _7);
(81)  (_7 < _8);
(82)  ((_7 + 1) == _8);
(83)  (_8 < _9);
(84)  ((_8 + 1) == _9);
(85)  (_9 < _10);
(86)  ((_9 + 1) == _10);
(87)  branch [_1, _2, _3, _4, _5, _6, _7, _8, _9, _10];
(88)  write "[" _1 ", " _2 ", " _3 ", " _4 ", " _5 ", " _6 ", "
      _7 ", " _8 ", " _9 ", " _10 "]";
```

The lines (68)–(86) define the constraints used when we lunch the command "example.". The line (87) shows what are the variables to search while the last line shows what information should be printed. The lines (68)–(87) are translated in Gecode in the following way:

```
(89)  IntVarArgs intVarArray2(10);
(90)  for(int int3= 0; int3<10; int3++)
(91)     intVarArray2[0+int3]= array[0 + int3 * 1];
(92)  dom(this, intVarArray2, 1, 10, opt.icl);
```

```
(93)   for(int int4= 0; int4<9; int4++) {
(94)      rel(this,array[0+int4*1],IRT_LE,array[1+int4*1],opt.icl);
(95)   }
(96)   for(int int5= 0; int5<9; int5++) {
(97)      IntVar intVar6(this, 1, 1);
(98)      IntVar intVar7=plus(this,array[0+int5*1],intVar6,opt.icl);
(99)      rel(this, intVar7, IRT_EQ, array[1+int5*1],opt.icl);
(100)  }
(101)  IntVarArgs intVarArray1(10);
(102)  for(int int2= 0; int2<10; int2++)
(103)     intVarArray1[0+int2]= array[0 + int2 * 1];
(104)  branch(this, intVarArray1, BVAR_SIZE_MIN, BVAL_MIN);
```

The lines (89)–(92) and (101)–(104) are used to define respectively the instructions defined in lines 68 and 87. The constraint with the path "((_i + 1) == _(i+1))" and "(i < _(i+1))" are defined using two "for" cycles in lines (93)–(95) and (96)–(100).

The programming languages used are the following ones:

- SICStus Prolog 4.0.1 is used for the SICStus to CNT tool
- c is used for the FlatZinc to CNT tool
- Haskell (`http://www.haskell.org`) is used for the CNT to Gecode tool

For the parsing tasks we also use the parsing generators Bison (`http://www.gnu.org/software/bison/`) and happy (`http://www.haskell.org/happy/`) and the lexical analyser generators Flex (`http://flex.sourceforge.net/`) and alex (`http://www.haskell.org/alex/`)

4 Experimental Results

We considered instances of four well-known problems, i.e. N-Queens, Sudoku, Golomb Rulers, and Knapsack.

Sudoku 16x16 instances are taken from `http://www.live-sudoku.com/play-online/geant`, and 25x25 ones are taken from `http://www.eleves.ens.fr/home/frisch/sudoku.html` ; instances for N-Queens problems range from $N = 100$ to $N = 115$; we launched Golomb rulers instances of order from 6 to 13, with two different lengths (the biggest satisfiable and the shorter unsatisfiable) for each order; knapsack instances are the same used in [8].

We modeled each problem in SICStus Prolog, MiniZinc, and Gecode. When available, we used the modeling offered by languages libraries. We also consider their translation with the tools described in the paper. We report the running times of the various versions in Table 2. The columns of the table are: *SICS*: pure SICStus Prolog 4.0.1 model; *SICS2GEC*: Gecode 1.3.1 model obtained compiling the SICStus Prolog model; *MZN2GEC*: Gecode 1.3.1 model obtained from the MiniZinc model compiled (by our tool) into a Gecode model; *MZN2FZNNI*: a FlatZinc model obtained from the MiniZinc model and run with the utility provided by NICTA research group; *MZN2FZNGE*: a FlatZinc model obtained from the MiniZinc model and run with the utility provided

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
16-0	0,05	0,01	0,01	75,68	594,27	0,07
16-1	0,00	0,00	0,00	0,32	0	0,00
16-2	0,12	0,06	0,03	1,24	20839,66	0,06
16-3	0,14	0,05	0,00	409,23	184,36	0,12
16-4	0,05	0,02	0,01	7,75	24,71	0,15
16-5	0,05	0,02	0,00	610,61	1204,99	0,07
16-6	0,10	0,03	0,04	24,16	666,97	0,08
16-7	0,89	0,55	0,27	12651	-	3,07
25-0	-	303206	164013	-	-	109799
25-1	-	669	879	-	-	395
25-2	-	52788	223	-	-	-
25-3	-	186362	57566	-	-	-
25-4	-	347	137	-	-	2547
25-5	-	89914	3560	-	-	-
25-6	-	44012	7123	-	-	-
25-7	-	147594	57620	-	-	-

Running times (s) for SUDOKU instances

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
100	0,13	0,87	0,02	4,59	-	0,00
101	60,67	0,95	14,03	25,81	-	3,83
102	0,16	1,41	0,02	5,13	-	0
103	0,13	468,73	0,02	5,64	-	0
104	0,12	0,14	0,06	6,00	-	0,02
105	8,71	186,42	1,93	8,89	-	0,55
106	0,22	35,79	0,06	5,72	-	0,01
107	0,2	0,19	0,09	6,26	-	0,02
108	2747	0,54	0,03	973,49	-	164,32
109	0,17	0,32	598,37	6,19	-	0
110	1704	0,15	403,67	801,28	-	119,78
111	0,34	0,18	0,22	7,56	-	0,05
112	0,21	93,11	0,08	7,32	-	0,02
113	0,43	-	0,08	6,97	-	0,02
114	0,17	3170	0,03	6,49	-	0,00
115	-	-	-	-	-	-

Running times (s) for N-QUEENS instances

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
6-sat	0,01	0,00	0,00	0,17	0,00	0,00
6-uns	0,00	0,00	0,00	0,18	0,01	0,00
7-sat	0,02	0,00	0,00	0,24	0,05	0,00
7-uns	0,06	0,01	0,02	0,23	0,09	0,00
8-sat	0,19	0,01	0,00	1,13	0,06	0,00
8-uns	0,67	0,13	0,18	0,92	0,27	0,03
9-sat	1,52	0,16	0,08	12,30	0,31	0,02
9-uns	5,96	1,26	2,19	10,66	1,87	0,24
10-sat	11,22	1,46	1,23	215,40	2,93	0,18
10-uns	46,73	10,51	26,70	166,81	27,01	1,95
11-sat	255,37	20,43	33,69	9404	294,05	0,91
11-uns	1406	316,77	1070	8342	1250	44,24
12-sat	2286	1181	9509	78616	5429	89,33
12-uns	8693	2134	16324	-	-	302,24
13-sat	56492	25560	-	-	-	1566
13-uns	-	-	-	-	-	6782

Running times (s) for GOLOMB RULERS instances

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
0-sat	0,01	0,01	0,00	0,17	0,01	0,01
1-uns	0,01	0,01	0,00	0,16	0,27	0,01
2-sat	0,16	0,05	0,05	0,21	0,30	0,60
3-uns	0,15	0,06	0,05	0,21	11,76	0,06
4-sat	3,74	1,28	1,23	1,05	13,37	1,27
5-uns	3,72	1,28	1,22	1,05	889,21	1,25
6-sat	156,88	53,35	51,08	26,89	1008,96	52,32
7-uns	155,41	53,42	50,58	25,65	123120	52,11

Running times (s) for KNAPSACK instances

– means that the execution didn't terminate after 4 days of execution time.

Table 2. Running times comparison of different encoding styles for various problems

by the Gecode Team that use Gecode 2.0 libraries; *Gecode*: pure Gecode 1.3.1 model. All codes and instances are available at [4].

There are two kinds of compile times: time of the compilation from high level code to Gecode C++ file and time needed by Gecode for internal compilation and libraries linking. With the proposed automatic detection of "`for`" loops both of them are rather low (the order of some seconds). Of course, for some small instances this time cannot be ignored w.r.t. execution time, but it becomes negligible for difficult instances.

Except for some 25x25 Sudoku instances, native Gecode code is always the fastest one, and this a is reasonable results, because native Gecode is the lower level way to encode the problems.

SICS2GEC often speeds-up SICStus native (because the SICStus is the higher-level encoding), except for some instances of N-Queens. Moreover it has, in average, comparable times with Gecode native code. Let us observe, however, that it solves all the Sudoku instances, while native Gecode does not.

The behavior of *MZN2GEC* is substantially equivalent with *SICS2GEC*, while in average it outperforms *MZN2FZNNI* and *MZN2FZNGE*, which are the standard ways to run MiniZinc (FlatZinc) models. Precisely, for Sudoku, N-Queens and Golomb rulers *MZN2GEC* is faster than *MZN2FZNNI* of various order of magnitude, while it is slightly slower in the case of Knapsack. We presume that the *MZN2GEC* performances are better than *MZN2FZNNI* and *MZN2FZNGE* ones, because when translating CNT models into C++ codes we perform precomputations and static analysis (see paragraph 3.4) that simplifies the variables domains and the set of constraints, w.r.t the execution of the flatzinc models. However, the utility provided by NICTA research group and the one of the Gecode Team directly execute the flatzinc files, without returning any intermediate encoding, so we can't perform an exhaustive comparison of the encodings.

We executed all tests on AMD Opteron 280 at 2.2GHz, Linux CentOS machine.

5 Conclusion and Future work

As a first step for the definition and implementation of a unified declarative modeling tool for combinatorial problems, we developed a compiler from SICStus and MiniZinc to Gecode. Although in its current preliminary version, is able to show that Constraint Programming can be done at high level using well-known languages and then executed in new paradigms as Gecode, since running times are comparable w.r.t. those of this latter paradigm. This way, we have ensured the feasibility of the three-phase constraint programming tool (modeling-translating-solving) we are developing. In fact, since the translating process can be done in reasonable time and the execution running times are comparable with native code ones, the user can benefit from both the flexibility and easiness of high-level modeling languages and the efficiency of the new low-level constraint solvers.

At this point a lot of work needs to be done. For instance:

- improving the static analysis of the generated code to further speed-up the overall process (compilation+execution)
- extending its compiling mechanism (up to now limited to CSP)
- porting the tool to the (new) Gecode 2.1.1, that should outperform the performances

– writing a front-end from CLP(FD) to FlatZinc: in this way, once we have the FlatZinc code of our model we can take advantage of the FlatZinc solvers

Moreover, as said above, this work is part of a more general project. We are currently working on hybridization between local search and constraint programming starting from Gecode and EasyLocal++ [6]. Basically, the solution's search of Gecode will be improved by this combined approach (in [3] we applied this hybrid approach to the protein structure prediction problem with interesting results). As a side effect, with the compiler described in this paper, one will be able, for instance, to program in Prolog and take automatically advantage of the hybrid search.

Acknowledgements The work is partially supported by MUR FIRB RBNE03B8KK and PRIN projects. We thank Luca Di Gaspero and Andrea Formisano for the useful discussions and the help in installing packages. We also thank Alberto Ghedin for releasing the first naive compiler from SICStus Prolog to Gecode.

References

[1] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. PLILP 1997:191–206

[2] R. Cipriano, L. Di Gaspero and A. Dovier. Hybrid Approaches for Rostering: A Case Study in the Integration of Constraint Programming and Local Search. *HM 2006*, LNCS 4030:110–123.

[3] R. Cipriano, A. Dal Palù and A. Dovier. A hybrid approach mixing local search and constraint programming applied to the protein structure prediction problem. *WCB 2008*. `http://wcb08.dimi.uniud.it/accepted.html`

[4] R. Cipriano, A. Dovier, and M. Jacopo. Tools for compiling SICStus and MiniZinc in Gecode. `http://www.dimi.uniud.it/dovier/MISIGE`

[5] A. Dal Palù, A. Dovier, and E. Pontelli. Heuristics, optimizations, and parallelism for protein structure prediction in CLP(FD). Proc. of PPDP 2005: 230-241, 2005.

[6] L. Di Gaspero and A. Schaerf. EasyLocal++: an object-oriented framework for the flexible design of local-search algorithms. *Software Practice and Experience*, 33(8):733–765, 2003.

[7] Daniel Diaz. GNU Prolog: a free Prolog compiler with constraint solving over finite domains. *Available from* `http://www.gprolog.org/`, 2007.

[8] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. *ICLP 2005*, LNCS 3668:67–82.

[9] A. Dovier, A. Formisano. and E. Pontelli. Multivalued Action Languages with Constraints in CLP(FD). *ICLP 2007* 4670:255–270.

[10] Gecode Team. FlatZinc/Gecode: a parser for FlatZinc modelling language. *Available from* `http://www.gecode.org/flatzinc.html`, 2007.

[11] Gecode Team. Gecode: Generic Constraint Development Environment. *Available from* `http://www.gecode.org`, 2006.

[12] J. Jaffar and J.-L. Lassez Constraint Logic Programming. *Tech. rep.*, Department of Computer Science, Monash University, June 1986.

[13] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[14] N. Nethercote. Specification of FlatZinc. Available from `http://www.g12.cs.mu.oz.au/minizinc/flatzinc-spec.pdf`.

[15] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. *CP 2007*, LNCS 4741:529–543.

[16] M. Wallace, S. Novello and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. *Technical report, IC-Parc, Imperial College, London*, 1997.

[17] Neng-Fa Zhou. B-Prolog: a versatile and efficient constraint logic programming (CLP) system. *Available from* `http://www.probp.com/`, 2006.

A SICStus Prolog codes

A.1 Sudoku model

```
sudoku(Cells) :-
   input(N,Cells),
   domain(Cells,1,N),
   row_constraints(Cells,N),
   column_constraints(Cells,N),
   square_constraints(Cells,N),
   labeling([ff,bisect],Cells).

%%% instance from www.repubblica.it, April 7 2008

input(9, [3,5,_,  _,_,_,  6,_,_,
          _,_,_,  8,_,_,  _,4,_,
          _,7,2,  _,_,5,  _,3,9,
          _,_,_,  1,9,_,  _,_,_,
          4,_,5,  _,_,_,  2,_,1,
          _,_,_,  _,6,2,  _,_,_,
          9,3,_,  4,_,_,  8,1,_,
          _,8,_,  _,_,1,  _,_,_,
          _,_,1,  _,_,_,  _,7,5]).


row_constraints([],_).
row_constraints(Cells,N) :-
   extract(N,Cells,Row,Rest),
   all_distinct(Row,[on(dom),consistency(global)]),
   row_constraints(Rest,N).

column_constraints(Cells,N):-
   column_constraints(Cells,1,N).
column_constraints(_,M,N) :-
   M > N, !.
column_constraints(Cells,M,N) :-
   column(Cells,M,N,Col),
   all_distinct(Col,[on(dom),consistency(global)]),
   M1 is M + 1,
   column_constraints(Cells,M1,N).

square_constraints(Cells,N):-
```

```
   M is integer(sqrt(N)),
   square_constraints(Cells,0,N,M).
square_constraints(_,N,N,_):-!.
square_constraints(Cells,I,N,M):-
   square(Cells,I,M,N),
   I1 is I + 1,
   square_constraints(Cells,I1,N,M).

square(Cells,I,M,N) :-
   Start is (I//M)*(N*M) + (I mod M)*M,
   extract(Start,Cells,_,MyCells),
   pick_square(MyCells,0,M,N,Square),
   all_distinct(Square,[on(dom),consistency(global)]).

pick_square( _,M,M,_,[]):-!.
pick_square(Cells,I,M,N,Square):-
   extract(M,Cells,Sq,_),
   (extract(N,Cells,_,Rest), !;
    true),
   I1 is I + 1,
   pick_square(Rest,I1,M,N,Uare),
   append(Sq,Uare,Square).

column([],_,_,[]).
column(Code,M,N,[A|R]) :-
   element(M,Code,A),
   extract(N,Code,_,RestCode),
   column(RestCode,M,N,R).

extract(N,Code,First,Last) :-
  length(First,N),
  append(First,Last,Code).
```

A.2 N-Queens model

```
queens(N, Queens) :-
    length(Queens, N),
    domain(Queens,1,N),
    constrain(Queens),
    labeling([ff], Queens).

constrain(Queens) :-
    all_distinct(Queens),
    diagonal(Queens).

diagonal([]).
diagonal([Q|Queens]) :-
    safe(Q, 1, Queens),
    diagonal(Queens).
```

```
safe( _,_,[]).
safe(X,D,[Q|Queens]) :-
    nonattacK(X,Q,D),
    D1 is D+1,
    safe(X,D1,Queens).

nonattacK(X,Y,D) :-
    X + D #\= Y,
    Y + D #\= X.
```

A.3 Golomb rulers model

```
golomb( Order, Length, Marks, Order ):-
    length( Marks, Order ),
    DiffNumber is (Order*(Order-1))//2,
    length( Diff, DiffNumber ),
    domain( Marks, 0, Length ),
    domain( Diff, 0, Length ),
    constrain_differences( Diff, Marks ),
    all_different( Diff ),
    break_simmetries( Marks ),
    labeling( [ffc], Marks ).
golomb( _, _, fail, _ ).

constrain_differences( Diff, [Hm|Tm] ):-
    constrain_mi( Diff, Hm, Tm, RemainingDiff ),
    constrain_differences( RemainingDiff, Tm ).
constrain_differences( [], _ ).

constrain_mi( [Hd|Td], Mi, [Mj|Tm], RemainingDiff ):-
    Hd #= Mj - Mi,
    constrain_mi( Td, Mi, Tm, RemainingDiff ).
constrain_mi( Td, _, [], Td ).

break_simmetries( Tm ):-
    Tm = [0|_],
    increasingMarks( Tm ),
    first_last_marks( Tm ).

increasingMarks( [ H1m , H2m | Tm ] ):-
    H1m #< H2m,
    increasingMarks( [ H2m | Tm ] ).
increasingMarks( [ _ ]).

first_last_marks( Marks ):-
    findFirsts( Marks, F1, F2 ),
    findLasts( Marks, L1, L2 ),
    F2 - F1 #< L2 - L1.

findFirsts( [F1,F2|_], F1, F2 ).
```

```
findLasts( Marks, L1, L2 ):-
    reverse( Marks, Rmarks),
    findFirsts( Rmarks, L2, L1).
```

A.4 Knapsack model

```
knapsack(Space,Profit) :-
    inputdata(Weights,Costs),
    length(Weights,N),
    length(Vars,N),
    domain(Vars,0,Space),
    scalar_product(Weights,Vars,#=<,Space),
    scalar_product(Costs,Vars,#>=,Profit),
    labeling([ff],Vars),
    write(Vars),nl.
```