From Zinc to Design Model

Reza Rafeh, Maria Garcia de la Banda, Kim Marriott, and Mark Wallace

Clayton School of IT, Monash University, Australia
{reza.rafeh,mbanda,marriott,wallace}@mail.csse.monash.edu.au

Abstract. We describe a preliminary implementation of the high-level modelling language Zinc. This language supports a modelling methodology in which the same Zinc model can be automatically mapped into different design models, thus allowing modellers to easily "plug and play" with different solving techniques and so choose the most appropriate for that problem. Currently, mappings to three very different design models based on constraint programming (CP), mixed integer programming (MIP) and local search are provided. Zinc is the first modelling language that we know of that supports such solver and technique-independent modelling. It does this by using an intermediate language called Flattened Zinc, and rewrite rules for transforming the Flattened Zinc model into one that is tailored to a particular solving technique.

1 Introduction

Solving combinatorial problems is a remarkably difficult task which requires the problem to be precisely formulated and efficiently solved. Even formulating the problem precisely is surprisingly difficult and typically requires many cycles of formulation and solving, while efficient solving often requires development of tailored algorithms which exploit the structure of the problem. Reflecting this discussion, recent approaches to solving combinatorial problems divide the task into two (hopefully simpler) steps. The first step is to develop the *conceptual model* of the problem which gives a declarative specification of the problem without consideration as to how to actually solve it. The second step is to *solve* the problem by mapping the conceptual model into an executable program called the *design model*. Ideally, the same conceptual model can be transformed into different design models, thus allowing modellers to easily "plug and play" with different solving techniques [8,6]. Here we describe the implementation of a new modelling language, Zinc [7], specifically designed to support this methodology.

We had three main aims when designing Zinc. First, we wanted the modelling language to be solver and technique independent, allowing the same conceptual model to be mapped to different solving techniques and solvers, i.e., be mapped to design models that use the most appropriate technique, be it local search, mathematical programming, constraint programming, or a combination of the above. Second, we wanted Zinc to provide high-level modelling features but still ensure that the models are executable. Thus, while Zinc provides sets, structured types, and user-defined predicates and functions, set domains must be finite and

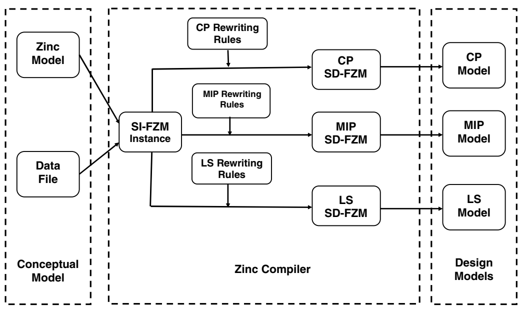

Fig. 1. Mapping a Zinc conceptual model to different decision models

recursion is restricted to iteration so as to ensure that evaluation terminates. And third, we wanted Zinc to have a simple, concise core that can be readily extended to different application areas by allowing Zinc users to define their own application specific library predicates, functions and types.

Of course there is considerable tension between these aims, since the higher-level the modeling language, the greater the gap between the conceptual model and the design model. The main contribution of this paper is to demonstrate that it is possible to map conceptual models written in a high-level modeling language, namely Zinc, into very different design models without introducing unnecessary overhead. This significantly extends our understanding of modeling language implementation since previous modelling languages and their implementations have been closely tied to specific underlying platforms and solving technologies. Note that, at this stage our objective is merely to minimise overhead, rather than competing with a directly encoded design model. In the future, we intend to build up a sufficiently broad range of transformations capable, under user control, of mapping a Zinc model to the best (known) possible design model.

Integral to the successful solver-independent implementation of Zinc is the use of an intermediate modelling language, called Flattened Zinc, to bridge the gap between conceptual and design model. Flattened Zinc is a subset of Zinc which is designed to be simple and low-level enough to be significantly closer to the decision model, yet sufficiently high-level to specify suitable intermediate models for all solvers. Therefore, it allows only simple constraints and data types.

The translation process from the conceptual model consisting of a Zinc model and instance specific data (optionally given in separate data files), to different design models is shown in Figure 1. The first step takes a Zinc model and performs syntax, semantics and type checking (which includes adding explicit coercions). The second step adds to the compiled Zinc model the information contained in the associated data file(s) (if any), and generates the Solver-Independent Flattened Zinc Model (SI-FZM) instance. This step is described more fully in Section 3. The advantage of first producing an SI-FZM model is that many common aspects of the mapping to the decision model can be performed during the Zinc to SI-FZM translation, thus reducing the burden when developing mappings to new solvers. In the third step rewrite rules are used to translate the SI-FZM into a Solver-Dependent Flattened Zinc Model (SD-FZM). As the name suggests, the rewrite rules used in this process depend on the target design model, and rewriting produces a Flattened Zinc model which is very close to the final design model. The mapping process is discussed in more detail in Section 4. The final step is to take the SD-FZM model and perform the minor syntactic rewriting required to generate the design model for a particular solving platform.

In our prototype implementation the Zinc model can be mapped into one of three design models, all of which are implemented in ECLiPSe. The first design model uses the standard constraint programming (CP) approach of a complete tree search with propagation based finite domain and set solvers. The second model is also complete but uses mathematical programming techniques, i.e. a Mixed Integer Programming (MIP) solver, while the third design model performs an incomplete search using local search methods. These are described and evaluated in Sections 5 and 6.

Modelling languages for specifying constrained optimization problems are one of the success stories of declarative programming. The first modelling languages, such as AMPL [4], provided little more than the ability to specify linear inequalities. More recent languages are considerably more expressive. Some are based on specification languages, e.g. ESRA [3] and ESSENCE [5], while others provide more programming language like features, e.g. OPL [13] and Localizer [10]. Zinc is somewhat similar to OPL but extends it by allowing constrained types and user-defined functions and predicates. The main innovation in Zinc is the ability to map a conceptual model to design models based on very different solving techniques. Other modelling languages have been designed for a particular underlying platform and solving technology. For example, AMPL is designed to interface to MIP solvers, ESSENCE is intended for propagation-based solvers, and Localizer was designed to map down to a local search engine. Although OPL models are automatically mapped to an underlying hybrid mathematical programming (MIP) and constraint programming library, the user cannot control the mapping to the same conceptual model of different design models. Also related is the mapping language Conjure [6], which uses rewrite rules to map ESSENCE models to an OPL-like language called ESSENCE'. The main difference is that while rewriting in Conjure produces alternative models for the same underlying solver, in Zinc it produces different models only for different solvers, tailoring the original model to the specific solver. Furthermore, to the best of our knowledge, a compiler for ESSENCE' has not been implemented yet.

2 Background: The Zinc Modelling Language

Zinc is a functional language with simple, declarative semantics. It provides: mathematical notation-like syntax (including automatic type coercions and operator overloading); expressive constraints (finite domain and integer, set and linear arithmetic); separation of data from model; high-level data structures and data encapsulation (including constrained types); user defined functions and

```
enum Customers ;
enum Products ;
array[Products] of set of Customers: Ordered ;

type Time = 1..card(Products) ;
array[Time] of var Products: Assign ;
array[Time] of var set of Customers: OpenStacks ;

constraint alldifferent([Assign[T] | T in Time]) ;

constraint forall(T in Time)
            (OpenStacks[T] ==
                allunion (Ti in 1..T) Ordered[Assign[Ti]]
                intersect
                allunion ( Ti in T..card(Products) ) Ordered[Assign[Ti]]);

minimize max([ card(OpenStacks[T]) | T in Time]) ;
```

Fig. 2. Zinc model for the Minimisation of Open Stacks Problem (MOSP)

constraints. We illustrate some of these features by means of a simple example. For more details the interested reader is referred to our earlier paper [7] which discusses the modelling capabilities of Zinc more fully.

Example 1. A Zinc model for the Minimisation of Open Stacks Problem (MOSP) is shown in Figure 2. In MOSP, a factory can manufacture a number of products but only one at a time. Once a product in a customer's order starts being manufactured, a stack is opened for that customer to store their products. Once all products for a customer are manufactured, the order is sent and the stack closed. The MOSP [15] aims at determining the time sequence in which products should be manufactured in order to minimise the maximum number of open stacks.

The first three lines of the model define the parameters: two enumerations `Customers` and `Products`, and an array `Ordered` indexed by `Product` containing the set of `Customers` who ordered that `Product`. Next, the two arrays of decision variables are declared where the `var` keyword is used to distinguish decision variables from parameters. The array `Assign` which assigns to each `Time` in the sequence a given `Product` to be manufactured, and the array `OpenStacks` which is constrained so that `OpenStacks[T]` is the set of `Customers` whose stacks are open at time `T`. The two following constraints indicate that (1) all products in array `Assign` must be different (i.e., each product is manufactured only once), and (2) the number of open stacks at time `T` is the intersection of those customers who ordered products manufactured before or at `T` and those who ordered products manufactured after or at `T`.

Data for the model can be given in a separate data file as, for example:

```
enum Customers = {C1, C2, C3, C4, C5};
enum Products = {P1, P2, P3, P4};
Ordered = [P1:{C1,C3,C5}, P2:{C2,C4}, P3:{C2,C3,C4}, P4:{C1,C5}];
```

List, set and array comprehensions provide the standard iteration constructs in Zinc. Other iterations such as `forall`, `max`, `allunion` and `sum` are defined as Zinc library functions based on the built-in function `foldl(F,L,Z)`, which applies the binary function `F` to each element in list `L` (working left-to-right) with the initial accumulator value set to `Z`. For example, the definition of `allunion` is

```
function var set of $T:allunion(list of var set of $T:L)=foldl(union(),L,{});
```

where `$T` is a type variable. Any constraint or function `F` (including user-defined functions or predicates) that takes a single list comprehension as an argument, can be called using the mathematical-like syntax `F(G) E`, which is equivalent to `F([E | G])`. Thus, for instance, `allunion (Ti in 1..T) Ordered[Assign[Ti]]` is syntactic sugar for `allunion([Ordered[Assign[Ti]] | Ti in 1..T])`.

One of the novel features of Zinc not illustrated in the previous example is that types can have an associated constraint on elements of that type. This generalises the idea of constrained objects [9] and allows to the modeller to specify the common characteristics that a class of items are expected to have. Two examples are:

```
type PosInt = (int:x where x>0);
record Activity = (var int: start, end, duration) where end=start+duration;
```

Zinc provides the standard comparison and equality operators, including the `alldifferent` constraint. These are polymorphic since all base types are totally ordered and overloaded versions of the operators are generated automatically for each user-defined type (using a lexicographic ordering for compound types).

Zinc allows constraints and variables to be annotated by classes which can contain attributes. These do not change the semantics of the model but can be used to guide generation of a decision model for a particular solver or solving technique. For instance, the annotation `penalty(p)` on a constraint indicates that with local search that constraint will be treated as a "soft" constraint with penalty p for violation.

3 Solver-Independent Flattened Zinc Model (SI-FZM)

As we have seen, Zinc is a very high-level, expressive modeling language. While this makes it ideal for developing conceptual models, it also introduces a considerable gap between the conceptual Zinc model and an associated design model targeted to a specific solver and search technique. The first step in bridging this gap is to translate the conceptual Zinc model into the Solver-Independent Flattened Zinc Model (SI-FZM). This is an intermediate representation oriented towards computer implementation, but still as solver-independent as possible. The SI-FZM is written in a subset of the Zinc language called Flattened Zinc which omits features of the Zinc model that make it user friendly, while preserving any features that could be used to support solver or search heuristics.

The first step to generate the SI-FZM instance from a Zinc model and its associated data file(s) is to insert all assignment statements from the data file(s) into the model. From then on, one or more of the following steps are performed to every statement in the problem instance:

- Evaluate all parameters and check the associated integrity constraints are satisfied.
- Determine an initial domain or range for all decision variables.
- Simplify record types by (a): replacing all records by tuples, (b) flattening tuples of tuples into a single tuple, and (c) appropriately replacing field access in the constraints by the contents of the field addressed.
- Replace enumerated types by integer range types, and constraints over enumerated types by the appropriate integer constraints.
- Check that predicates and functions are sufficiently instantiated. For example, `foldl` requires its second argument to be a list of known length.
- Unfold the user-defined library and built-in predicates and functions such as `foldl`. Note that this may introduce new variables due to the formal parameters and to the existence of local variables in the definitions.
- Insert constraints arising from constrained objects, i.e., from the constraints associated with types. If these involve only parameters, check that they hold.
- Simplify arrays and lists by rewriting them to be one-dimensional arrays with an integer index set starting from 1, and appropriately updating computation of the array index in constraints.
- Translate variable sets of structured types into variable sets over integers and add a constraint mapping the structured type elements to integers. This is also used to flatten sets of sets into linked sets of integers. For instance:

  ```
  var set of {{2,5},{1,3,6},{1,2}}: S1;
  var set of {{2,5},{1,2},{3,4}}: S2;
  constraint S1 intersect S2 == {1,2};
  ```

  is translated to (assuming the encoding starts from 1):

  ```
  var set of {1,2,3}: S1;           var set of {1,3,4}: S2;
  constraint S1 intersect S2 == 3;
  ```

- Separate the logical combination of constraints from the constraints themselves, using reification, i.e., substituting c by `reify`(c, T) which constrains Boolean variable T to be `true` iff c holds. For example, the constraint $c \equiv (x < y \vee x < z) \wedge (x > w)$ is substituted by:

  ```
  constraint reify(x < y , T1);      constraint reify(x < z ,T2);
  constraint reify(x > w , T3);      constraint T4 = T1 \/ T2 ;
  constraint T4 /\ T3 ;
  ```

  Note that reification is performed after unfolding predicates and functions, leaving only constraints defined by the underlying solvers. For constraints whose reification - and more specifically negation - is not supported by the solver, e.g. linear constraints in continuous variables, the reification is implemented using a specific transformation (in this case adding an ϵ so $\neg X \geq Y$ is transformed to $X \leq Y - \epsilon$)

Termination of the Zinc flattening is guaranteed as long as the unfolding of predicates, functions and iterators terminates, and only finitely many new variables are introduced. These conditions are guaranteed by the Zinc syntax.

We will illustrate some of these operations using the Zinc MOSP model given in Figure 2 with the data file of Example 1. The arrays `Assign` and `OpenStacks` which mapped `Time` to `Products` and to `set of Customers`, respectively, are translated into the FZM code:

```
array[{1, 2, 3, 4}] of var 1..4 : Assign ;
array[{1, 2, 3, 4}] of var set of 1..5 : OpenStacks ;
```

where the index type `Time` has been replaced by its range value `1..4` (represented using the more general set `{1,2,3,4}`), and the enumerated types `Products` and `Customers` have been replaced by ranges 1..4 and 1..5, respectively. Next, the `alldifferent([Assign[T] | T in Time])` constraint is translated as:

```
constraint alldifferent([Assign[1], Assign[2], Assign[3], Assign[4]]);
```

and the `forall` constraint is unfolded to give the four `OpenStacks` elements:

```
constraint
    OpenStacks[1]==(T_1) intersect (T_1 union T_2 union T_3 union T_4)
 /\ OpenStacks[2]==(T_1 union T_2) intersect (T_2 union T_3 union T_4)
 /\ OpenStacks[3]==(T_1 union T_2 union T_3) intersect (T_3 union T_4)
 /\ OpenStacks[4]==(T_1 union T_2 union T_3 union T_4) intersect (T_4);
```

where each temporary variable `T_i` is equated to the result of the expression `Ordered[Assign[i]]`. Thus if `Assign[i]` = 1 then `T_i` = `{1,3,5}`, and if `Assign[i]` = 2 then `T_i` = `{2,4}`, etc. This is expressed using the standard constraint programming global constraint `element(I,L,X)` which holds if `X` is the `I`th element in `L`, i.e. `X = L[I]`. The specific SI-FZM constraint is:

```
 constraint element(Assign[i], [{1,3,5}, {2,4}, {2,3,4},{1,5}],T_i);
```

The flattened Zinc `element` constraint allows lists of complex types, rather than only the usual lists of integers.

Some of the optimisations used to improve the generated SI-FZM model are:

1. Substitution: If we can determine that a decision variable must take a unique value, then we can effectively treat it as a parameter and replace it by its value. For example, if we know that $X == 2$, constraint $X \times Y \geq 10$ can be simplified to $2 \times Y \geq 10$.
2. Omitting unnecessary `element` constraints: While Zinc supports arrays with arbitrary index sets, the `element` constraint supported by most solvers requires a range of the form $1..n$ as its index set. Thus, when we model an array access we use an extra `element` constraint to map the index set variable to a range of the required form. For example, the constraint in the Zinc code:
   ```
   array[{2,5,7,8}] of var int:A;
   var {2,5,7,8}:I;
   constraint A[I]==3;
   ```
   generates the Flattened Zinc code:
   ```
   var 1..4:T_1;          var int:T_2;
   constraint element(T_1,[2,5,7,8],I);
   constraint element(T_1,[A[2],A[5],A[7],A[8]],T_2);
   constraint T_2==3 ;
   ```

However, if the index set of the initial Zinc array is in fact a range, then we can replace the extra `element` constraint by an offset to the index variable. For instance, if we have `array[4,5,6,7] of var int:B` we can substitute `B[J]` by a new variable `T` which is constrained by a single `element` constraint:

```
constraint element(J-3,[B[4],B[5],B[6],B[7]],T);
```

3. Simplifying reifications and omitting unnecessary reification: While the naive translation of compound constraints of the form `constraint C1` $\wedge$ C2 is

```
var bool: B1, B2;            constraint reify(C1,B1);
constraint reify(C2,B2);     constraint B1 /\ B2;
```

it is better to produce the simpler code

```
 constraint C1;               constraint C2;
```

which removes the potential overhead of reification and is more efficient, especially for MIP techniques.

One source of inefficiency in Zinc is the current lack of common sub-expression elimination for constraints which appear several times in our models. As a result, multiple `element` and/or `reify` constraints are created, instead of reusing the associated variables. We are currently resolving this issue.

4 Model to Model Transformation

Although the SI-FZM model is much closer to a design model than the original Zinc model, it may still contain constraints and data structures not supported by the intended solver. For example, Zinc supports variable sets of any type. Since current set solvers can support only sets over integer values, variable sets in a Zinc model are transformed to variable sets over integers in the generated SI-FZM. For the many solvers, including most MIP solvers, that do not support sets of integers, integer sets must in turn be converted to some other representation they can handle, such as Boolean arrays. To facilitate this kind of transformation the Zinc implementation supports solver specific rewrite rules that can be used to rewrite the SI-FZM model to a Flattened Zinc model that is much closer to the desired design model. Rewrite rules have the following syntax:

```
if A then substitute B with C in D where E;
```

where A is a conditional statement, B and C are two Zinc expressions, D is a subsection of a Zinc model (*declarations*, *constraints* or *model*) and E is a set of Zinc statements. Whenever A holds, all instances of B are substituted by C in scope D and the statements in E are added to the model. The *if* and *where* parts are optional.

The formal semantics of our rewrite rules is not yet fully worked out. A key issue is the specification of what can be tested in a conditional statement. In the rules used to date, the conditions have been restricted to tests on Zinc types.

Example 2. Consider the following four rules, which are among those used in our implementation to map a set S to an array of Boolean variables B, such that $B[x] \leftrightarrow x \in S$.

```
(1) substitute var set of $T:X with array[domain(X)] of var bool:X
      in declarations;

(2) if typeof(X)==array[$T] of var bool then
         substitute (I in X) with
         (if I in indexset(X,1) then X[I] else false)
          in constraints;

(3) if typeof(X)==set of $T then
        substitute X  with Z in constraints
        where
            array[$T] of bool:Z=[I:true| I in X];

(4) if typeof(L)==list of array[$T] of var bool then
        substitute element(I,L,X)
        with element(I, [extend(L[K],U)|K in 1..length(L)], extend(X,U))
        in constraints
        where U = unionall({indexset(L[H],1) | H in 1..length(L)})
                  union indexset(X,1);
```

and function `extend` is defined as:

```
function array[$T] of var bool: extend(array[$T] of var bool:B, set of $T:U)
         = [ if J in indexset(B,1) then B[J] else false | J in U]
```

Rule (1) substitutes in every declaration, any variable set `X` of some type `$T` by a Boolean array, assuming all set constraints can be mapped to equivalent constraints on Boolean arrays. Rule (2) rewrites the set membership expression `(I in X)` for any `X` known to be a Boolean array as a result from previous rule, into the expression `X[I]`.[1] Zinc keeps track of which expressions have been newly introduced as a result of the mapping. Rule (3) maps the set of values `X` of some type `$T` into a Boolean array `Z` in which every element `I in X` is assigned value `true`. Rule (4) is used for an `element(I,L,X)` constraint in which `L` was a list of set values that has been transformed by Rule (3) into a Boolean array. It transforms the constraint into another `element` constraint whose second argument is a list of Boolean arrays, each defined over the same index set, U. Function `extend` extends an array of Booleans to a larger index set U, by adding the Boolean value false for each new index. It returns an array of Booleans over the extended index set U. The `extend` function has been used for readability reasons; in the implementation of Rule (4), the function is already unfolded. If we apply the above rules to the following code from the generated SI-FZM for the Open Stack Problem discussed in Section 3:

```
var set of {1,2,3,4,5} : T_3;
constraint element(Assign[1],[{1,3,5}, {2,4}, {2,3,4}, {1,5}], T_3);
```

the SD-FZM would be generated as follows (`t` stands for *true* and `f` for $false$):

```
array[{1,2,3,4,5}] of var bool : T_3;
constraint element(Assign[1],[[t,f,t,f,t],[f,t,f,t,f],
                          [f,t,t,t,f],[t,f,f,f,t],T_3);
```

[1] $indexset(A, I)$ returns the index set of the I^{th} dimension of array A.

First, Rule (1) changes the definition of T_3 in the declaration section. Then, Rule (3) maps all sets in the constraint into Boolean arrays. Finally, Rule (4) makes the length of all Boolean arrays equivalent by adding $false$ for each missing member from set 1..5.

Our current implementation uses 20 rewriting rules of which 16 are used for transforming set constraints to constraints over Boolean arrays, and the remainder are used for implementing suitable versions of `max`, `min`, `maxlist` and `minlist` constraints in MIP techniques.

5 Mapping to Design Models

The primary focus of this paper is to investigate whether the high-level modelling language Zinc can provide solver and technique independent modelling. To do so, we must demonstrate that it is possible to map SI-FZM to design models using different solving techniques, and that the resulting design models do not suffer substantial overhead as compared to equivalent design models written by hand. To investigate this we have implemented mappings from SI-FZM to three very different design models.

For practical reasons all three design models were implemented using the ECLiPSe system [1]. We see no apparent reason why the choice of system should impact our experiments concerning the mapping overhead.

Mapping to CP: The SI-FZM constraints are mapped to finite domain propagation constraints. A simple complete tree search using variable *labeling* is added, and the CP system solves the problem using search and propagation. Standard CP propagation solvers typically support the SI-FZM constraints such as `reify`, `>=`, `=\=` etc. Specifically, we have used the ECLiPSe solvers `ic`, `ic_sets`, `ic_global` and, to support search and optimisation, the ECLiPSE `branch_and_bound` library.

We extended these libraries to provide comparison operators on compound data objects by generating a new constraint for each comparison operator and type. For example, the constraint `[a1,a2] =< [b1,b2]` effectively generates

```
(a1 < b1) \/  ((a1=b1) /\ (a2 =<  b2))
```

We also implemented a more general `element` constraint since, like most CP systems, ECLiPSe provides only a restricted form of `element` constraint which requires the list argument to be a ground list of integers. This more general version of `element` delays evaluation until two of its arguments are fixed.

Mapping to MIP: The SI-FZM constraints are mapped to integer and linear numeric constraints, and the problem is solved using standard MIP branch and bound search. This mapping is considerably more complex because the class of constraints handled by MIP is much more restricted.

Set constraints are mapped to Boolean constraints, which are in turn mapped to constraints over binary integer variables as detailed in Section 4. The remaining SI-FZM constraints are handled by specific translations.

Reified constraints are translated using the *Big M* technique [14]. For instance, if we assume X and Y are numeric variables and B is a binary integer variable, we model $reify(X \leq Y, B)$ by the inequalities

$$X + B \times M \leq Y + M \wedge X + M \geq Y + (1 - B) \times M + \epsilon$$

where M is a big number and ϵ a small number. If B becomes 0, the first constraint is relaxed while the second constraint forces X to be greater than Y. Otherwise, if B becomes 1, the first constraint forces X to be less than or equal to Y while the second constraint is relaxed.

Some global constraints, such as `alldifferent` have a standard mapping to MIP, as introduced in [11]. More novel and interesting is the mapping of the *element* constraint. For efficiency the translation depends on how the arguments of the constraint $element(I, L, X)$ are instantiated.

- I is instantiated to the value i: the translator impose an equality constraint between X and the i^{th} element of L.
- L and X are ground: the translator finds the set of positions $S = \{i : L[i] = X\}$. The constraint is then translated as `var S : I`.
- Only L is ground: We associate a binary integer variable with each member of L. For each member Y, if $X = Y$, its associated binary variable becomes 1, otherwise 0. Assuming $L = [a_1, a_2, ..., a_n]$, the constraint `element(I,L,X)` is converted to the following constraints:
  $b_1, b_2, ..., b_n :: 0..1$, $integers([b_1, b_2, ..., b_n])$, $\sum_{i=1}^{n} b_i = 1$,
  $I = \sum_{i=1}^{n} i \times b_i$, $X = \sum_{i=1}^{n} a_i \times b_i$
  The first constraint restricts the range of each variable b_i to 0..1, the second enforces integrality, so $b_i \in 0, 1$, and the third checks that only one of the n binary variables is non-zero. On the next line, the fourth and fifth constraints establish the relationship between binary variables and I and X, respectively.
- Otherwise, in the case that L is not completely ground we use the above translation except that for each non-ground a_i, instead of generating the constraint $X = \sum_{i=1}^{n} a_i \times b_i$ we generate the two constraints: $X - M \leq a_i - M.b_i$, $X + M \geq a_i + M.b_i$, where M is a sufficiently large number. These behave like the *Big M* technique used for handling reification.

Mapping to local search solver: The final mapping uses a form of local search. Annotations on the constraints in the original Zinc model guide which constraints are enforced, i.e. hard, and which are handled by using a penalty in an automatically generated objective function, i.e. treated as soft.

The local search algorithm used for the experiments described in the next section is a hill-climber, with a tabu facility to prevent cycling on a plateau. The algorithm selects a variable in conflict, if there is one, and otherwise any variable. The value of the variable is changed and the algorithm then completes the move by changing any other variable values that are required by the hard constraints. The completion is greedy in the sense that each choice of variable and new value generates only one move. The neighbourhood search first considers integer variables generated from Zinc model variables, and then set variables, generated

Table 1. Zinc Mapping Statistics

Problem Name	Model Size		Generated Model Size			Mapping Time (sec)		
	Zinc	ECLiPSe	SI-FZM	SD-FZM	ECLiPSe	SI-FZM	SD-FZM	ECLiPSe
Golfers (sets)	273/5	-	1082	10706	65720/2492	20.119	31.262	34.143
Golfers (arrays)	269/5	1111/5	67451	43684	17178/485	0.1	0.475	0.55
Job-Shop	514/3	1021/5	9980	9980	16634/574	1.564	1.589	1.673
Knapsack	326/1	564/3	896	589	1181/2	0.2060	0.2220	0.2250
Stable-Marriage	527/4	955/4	16064	16064	24604/672	0.4770	2.3280	2.4330
Queens	88/3	308/3	81	81	245/3	0.047	0.047	0.047
Open-stacks	264/2	723/5	5240	29649	5104/981	0.1530	1.1070	1.4250
Perfect-squares	322/3	630/6	23446	23368	43289/231	1.578	1.644	1.933
Production	173/2	367/3	173	173	173/6	0.4190	0.4250	0.4480

from Zinc set variables. Auxiliary variables, introduced during the mapping, are automatically updated by the local search, via the introduced hard constraints that relate them to the original variables.

6 Evaluation

Our primary motivation for developing Zinc was to validate the idea of a high-level modelling language which is solver and technique independent. Therefore, our evaluation aims at demonstrating two things. First, that it is possible to map Zinc models to design models using different solving techniques. And second, that the resulting design model does not suffer substantial overhead when compared to an equivalent design model written by hand To achieve this, we used as benchmarks the Zinc model for the MOSP problem (9 customers, 7 products) given in Figure 1, and models for the following well known problems:

– Perfect Squares (7x7, 14 squares) - because of its use of disjunction
– Queens (18 queenss) - it spawns a large number of constraints.
– Knapsack (30 objects, 50%fit) - it has sets with multiple constraints on them.
– Stable Marriage (8 pairs) - it uses arrays with variable indices.
– Social Golfers (6 players, 3 groups, 3 weeks) - it uses sets of sets.
– Social Golfers (flat sets) - to reveal the cost of supporting sets of sets.
– Job Shop (4 jobs, 3 machines) - it uses many modelling features of Zinc.
– Production (3 products, 2 resources) - it involves continuous variables.

Our prototype implements the full syntax of Zinc. It is written in Mercury with a Yacc generated parser and flex generated lexical analyser. It is about twelve thousand lines of Mercury code, and five thousand lines of C. Experiments were performed on a 3GHz Pentium 4 with 1Gb memory on Fedora.

Table 1 gives statistics on the mapping using MIP techniques. The results for the other two mappings are similar, just a little bit smaller because MIP techniques cannot support high-level constraints and must be mapped to simpler ones. The first five columns give the size of the models as *number of "tokens" /*

number of constraints for the original Zinc model (in addition to the data file), the direct ECLiPSe program, and the generated SI-FZM, SD-FZM and ECLiPSe model, respectively. The last three give the time in seconds taken to generate the SI-FZM, SD-FZM and ECLiPSe model, respectively. Note that we do not give a model written directly in ECLiPSe for Golfers (sets), since it is not naturally expressible in ECLiPSe.

The Zinc model is consistently substantially smaller than the model written directly in ECLiPSe. The SI-FZM and generated ECLiPSe code is orders of magnitude larger than both the Zinc model and the direct ECLiPSe model. This is to be expected and reflects the flattening of high-level iteration constraints. Thus, the size is proportional to the number of constraints sent to the solver rather than to the number of constraints in the original model. The time to generate the ECLiPSe design model from the Zinc model is small, no more than a few seconds, for all mappings and examples, except for Golfers (sets), due to the number of set-related constraints generated, which grows exponentially. We are currently studying how to tackle this issue.

Our second experiment aimed at determining if the ECLiPSe code generated from the Zinc model had a substantial overhead as compared to an equivalent model written directly in ECLiPSe. Thus, we compared their execution times for all three design models: Constraint Programming (CP), Local Search (LS) and MIP. Table 2 shows the execution time in seconds for all programs when finding the first solution.

One possible confounding factor is the choice of search strategy. Clearly, this can greatly effect the performance of the design model. Since we are only interested on the relative performance of the two models, we ensured (as far as possible) that the direct ECLiPSe model used the same search strategy as that in the generated model. This is the reason behind the differences in the execution time for the two MIP models for Queens and Perfect Squares which, despite our efforts, perform different searches and return different solutions. Note that there were three problems whose structure was too complex to be solved with reasonable efficiency with our generic "blind" local search algorithm. These are indicated as "-" in the table.

Table 2. Comparing the execution times for the direct and mapped programs

Problem Name	CP Model		MIP Model		LS Model	
(cpu secs)	Direct	Generated	Direct	Generated	Direct	Generated
Golfers (sets)	-	0.343	-	1.34	-	0.156
Golfers (arrays)	0.031	0.0	0.172	0.266	0.0	0.0
Job-Shop	0.094	0.109	4.125	3.218	0.375	0.39
Knapsack	22.828	22.675	0.00	0.00	0.797	0.828
Stable-Marriage	0.031	0.031	3.391	3.047	-	-
Queens	4.125	4.109	8.64	24.094	11.61	11.641
Open-stacks	1.547	1.843	1890.797	1971.688	0.94	0.95
Perfect-squares	0.031	0.031	3.469	1.5	-	-
Production	33.328	33.188	0.00	0.00	-	-

Table 2 shows no significant difference in execution time between the design model written directly in ECLiPSe and that generated from the Zinc model. This is true for all the design models: CP, LS and MIP. This preliminary evidence encourages our pursuit of a high level, solver independent modelling language.

7 Conclusion

We have presented the implementation of the first prototype of the modelling language Zinc. Unlike virtually all other modelling languages, a Zinc model can be mapped into design models that utilize different solving techniques such as local search, tree-search with propagation based solvers, or MIP techniques. A core feature of the Zinc implementation supporting such solver and technique-independent modelling is the use of an intermediate language called Flattened Zinc. Furthermore, the Zinc implementation provides a rewrite rule based model to model transformation facility to allow the implementers to map the Flattened Zinc model into one that is closer to the desired technique/solver.

We have compared a number of standard benchmarks written in Zinc and written in ECLiPSe. The Zinc models are considerably more concise and arguably more high-level and easier to understand. The ECLiPSe model automatically generated from Zinc (via FZM) has similar performance to an equivalent program written in ECLiPSe, assuming the same search method is used for all three mappings. This provides strong support for the hypothesis that it is possible to generate reasonably efficient design models from Zinc, and so allow Zinc modellers to readily experiment with different solving techniques. For instance, it is clear from our experiments that for the `Knapsack` and `Production` benchmarks MIP is the better technique, while for the others the CP propagation solver is the best. In the future, we plan to experiment with hybrid techniques.

Zinc has been developed as part of the G12 project and is intended to be its modelling language. Currently, mappings from Zinc to the three different design models have been crafted in Mercury with some transformations using rewrite rules. Besides the ECLiPSe platform, Zinc models will also be mapped down to Mercury itself [12]. In the longer term, we plan to use a specialised term rewriting language (Cadmium [2]) to implement the mappings from Zinc to Flattened Zinc along with model-transformations.

An important component of the mapping from conceptual to decision model is specification of the search. Currently, our implementation uses a naive search procedure, but user-controlled search is vital for scalable performance on real problems. Specification of search is deliberately not part of the Zinc language, since we believe this should not be part of the conceptual model. However, search is often naturally specified in terms of the variables and entities occurring in the decision model, so it seems sensible to allow the search component to be written in a Zinc-like language annotating the Zinc model. The inability to specify problem specific search is almost certainly the reason that the local search mapping was not competitive. We are currently exploring this.

Acknowledgements. We also like to thank members of the G12 team at National ICT Australia for helpful discussions, in particular Ralph Becket, Nick Nethercote and Peter Stuckey.

References

1. K. Apt and M. Wallace. *Constraint Logic Programming Using ECLiPSe.* Cambridge University Press, 2006.
2. G. Duck, P.J. Stuckey, and S. Brand. ACD term rewriting. In S. Etalle and M. Truszczynski, editors, *Proceedings of the International Conference on Logic Programming*, number 4079 in LNCS, pages 117–131. Springer-Verlag, August 2006.
3. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR*, pages 214–232, 2003.
4. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2002.
5. A.M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The essence of ESSENCE: A constraint language for specifying combinatorial problems. In *Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 73–88, 2005.
6. A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The rules of constraint modelling. In *Proc 19th IJCAI*, pages 109–116, 2005.
7. M. Garcia de la Banda, K. Marriott, R. Rafeh, and M. Wallace. The modelling language Zinc. In *Proc. CP06*, pages 700–705. Springer-Verlag, 2006.
8. C. Gervet. *Large scale combinatorial optimization: A methodological viewpoint*, volume 57 of *Discrete Mathematics and Theoretical Computer Science*, page 151ff. DIMACS, 2001.
9. B. Jayaraman and P. Tambay. Modeling engineering structures with constrained objects. In *PADL*, pages 28–46, 2002.
10. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Proc. Principles and Practice of Constraint Programming - CP97*, pages 237–251, 1997.
11. P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In *CP*, pages 369–383, 2000.
12. P. J. Stuckey, M. J. García de la Banda, M. J. Maher, K. Marriott, J. K. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *CP*, pages 13–16, 2005.
13. P. Van Hentenryck, I. Lustig, L.A. Michel, and J.-F. Puget. *The OPL Optimization Programming Language.* MIT Press, 1999.
14. H. P. Williams. *Model Building in Mathematical Programming.* John Wiley and Sons Ltd, 1999.
15. B. J. Yuen and K. V. Richardson. Establishing the optimality of sequencing heuristics for cutting stock problems. *European Journal of Operational Research*, 84:590–598, 1995.