The Mozart Constraint Subsystem

System Presentation

G. Gutiérrez[1], A. F. Barco[2], M. A. Villanueva[3], A. Cardona[2], D. Montenegro[2], S. Miller[2], J. F. Díaz[3], C. Rueda[2], and P. Van Roy[1]

[1] ICTEAM, Université catholique de Louvain
[2] Dept. of Electronics and Computer Science, Pontificia Universidad Javeriana-Cali
[3] School of System Engineering and Computer Science, Universidad del Valle
Corresponding author: `gustavo.ggutierrez@gmail.com`

Abstract. We present the current state of a new implementation of the constraint engine for the Mozart programming system. Our implementation integrates the Gecode constraint library into the core of Mozart version 2.0. Doing so, we allow users to take advantage of the efficiency of Gecode propagators transparently by maintaining the existing language constraints abstractions. Future Mozart systems can thus benefit from the rapid pace of constraint solving optimizations that are included in each new Gecode version. We use two well-known puzzles to illustrate the system and present all available abstractions.

1 Introduction

Constraint programming has been identified as a key area in the expansion of applied computer science. Several layers of abstractions become interesting in the solution of combinatorial problems and a clear programming language for modeling along with an state of the art implementation of constraints services is highly desired. On this regard, we present the integration of a new implementation of constraint engine into the Mozart Programming System version 2.0. In essence, the Gecode library[1] substitutes the old constraint engine of Mozart, providing all the mechanisms to solve Constraint Satisfaction Problems (CSPs) as do the previous Mozart interfaces.

Several aspects of the implementation need to be taken into account for this integration to be successful. Most of them come from constructs that are possible in Mozart and that require interaction with the constraint subsystem:

- Garbage collection of constraint variables
- Implementation of branching heuristics in Oz that will be used by the Gecode search engines
- Definition of search engines directly in Oz
- Interaction between threads and the constraint subsystem
- Support for programming paradigms like logic programming

The first goal of this integration is to provide an efficient abstraction for constraint programming in Mozart that uses the Gecode constraint library. This basically means being able to state and solve CSPs in the new Mozart implementation. At this stage CSP solving does not include all the Gecode functionality and the support for all the Oz constructs will be limited.

Interestingly enough this goal has a lot of difficulty. It fixes the design in which important components like the Mozart virtual machine interacts with Gecode. It also defines the representation and interaction of constraint variables in the virtual machine and their corresponding Gecode variables. Moreover, it has to define how constraint variables interact with other variables and language constructs.

After achieving this first goal we plan to focus on making all Gecode functionalities available to the Oz user. This includes, for instance, the use of predefined search engines and constraint techniques such as efficient propagation and advanced state restoration techniques [2, 5, 6].

In this paper we describe the initial view of the system and what has to be done in order to have a full integration and a backward compatible version of Mozart. We also take this as an opportunity to announce the alpha version of the constraint subsystem of Mozart 2.0.

The structure of the paper is as follows. Section 2 presents the basics for the general interaction between the two systems. Deeper insight of that integration is presented in the subsequent sections. Section 3 shows how constraint variables are defined in Oz and how them interact with their corresponding Gecode implementations. Posting constraints is explained in section 4 and the interaction between Mozart and Gecode to perform constraint propagation is explained in section 5. Sections 6 and 7 present how branchers and search engines work in the new integration. Next, section 8 describes all available abstractions showing the code of two well-known puzzles. Finally, sections 9 and 9.1 presents some conclusions and future work.

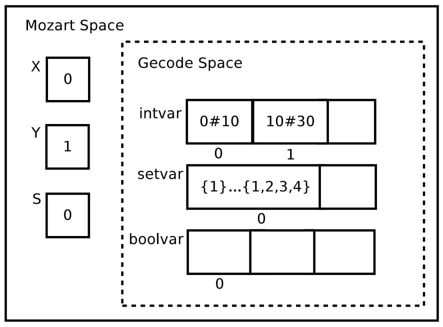
2 Mozart constraint subsystem design

In order to *delegate* the constraint programming support in Mozart to Gecode we first describe the relevant design similarities between both systems and then define the level and the design of the interaction between the systems.

Mozart and Gecode are in essence very similar systems in which regards to constraint programming. Both of them use the notion of computation spaces(or just spaces)[4, 3] to encapsulate constraint variables and propagators. This similarity is reflected in Mozart at the design level by having an explicit implementation for spaces. This is then the first and most natural entry point of the integration: every mozart space is associated with a corresponding gecode space. The idea is then that the gecode space encapsulates all the data and functionality of the constraint programming related operations.

This is implemented by adding to the mozart space class a gecode space as an attribute. Such attribute cannot be just a raw gecode space but is instead a pointer for reasons that we discuss in section 3. We thus use a custom class that inherits from `Space`[4].

Figure 1 presents a simple CSP problem written in Oz and its corresponding representation in the Mozart virtual machine. The details and design decisions concerning this proposal are discussed in the following sections.



```
proc {CSP R}
  X :: 0#10    Y :: 10#30
  S = {FS.dom {1} {1,2,3,4}}
in
  R = [X Y S]
  % Some constraints
  % Distribution
end

{Space.new CSP Sp}
```

Fig. 1: Low level representation for the Mozart space created by the program on the right.

3 Constraint variable declaration

Variable declaration is a basic functionality in constraint programming. To support it we have to consider the interaction between the Mozart virtual machine, the Mozart space in which the variable is declared and its corresponding Gecode space where the variable is represented.

The following actions are performed when declaring a constraint decision variable.

– A new variable in the Mozart space is created
– A new variable in the Gecode space is introduced
– An association between the two is maintained

Notice that at this point we are not concerned with the type of constraint variable. The actions described above are independent of the variable type. Moreover, any interaction between the variables in the gecode space will be performed through the correspondent Mozart variables.

As we do not know in advance how many constraint variables will be required we need to provide a mechanism to add constraint variables to the gecode space. For this we supply each gecode space with one vector for each constraint variable type. Those vectors correspond to `intvar`, `setvar` and `boolvar` depicted in fig. 1.

[4] Class name implementing the notion of computation space in Gecode

A constraint variable in Mozart is declared by calling a function that has a C++implementation. This function gets parameters according to the variable type. For instance, the function to define a finite domain variable takes two integers for the minimum and maximum values. Internally, a new constraint variable is created in the gecode space and initialized accordingly. A new variable is also created in the Mozart virtual machine that can be used to access the Gecode variable. To link the two variables we store in the Mozart variable the index of the constraint variable in the corresponding vector of the gecode space.

The association between Mozart and Gecode variables is one-way only. We have found that this functionality is enough because, up to this time, we have not found the need to interact with Mozart variables from Gecode. For propagator posting and variable access operations this one-way interaction suffices.

To display information about constraint variables in Mozart we require an extra operation that provides an Oz representation of the constraint domain of the variable. This operation, in the case of finite domains, will construct a list of integers (resp. pairs) with every element (resp. range) of the variable domain. For finite sets domains the operation constructs two list of integers with the lower and upper bounds of the corresponding set variable.

4 Propagator posting

Posting a constraint in Oz is done (as before) by calling a suitable procedure. The arguments of this procedure are the variables the constraint will be posted on together with some additional information. For instance, consider the simple example of posting constraint $X+Y=5$. Assuming that X and Y were declared as finite domain variables, this constraint is posted by calling the procedure `{FD.linear post([1 1] [X Y] '=:' 5)}`.

The implementation of `FD.linear`, written in C++, performs the following operations:

1. For the list of coefficients `[1 1]` and the constant `5` create type compatible Gecode counterparts. For instance, for the list of integers it creates an `IntArgs` and for the constant an integer.
2. For the list of variables `[X Y]` create references, say X_g and Y_g, to the associated Gecode variables. These variables are stored in an `IntVarArgs`.
3. For the atom `'=:'` create the correspondent integer relation type value.
4. After all the information is extracted and represented in a Gecode compatible way post the constraint in the gecode space. The code for this is: `linear(gspace,iva,ia,irt,c);`. Where `gspace` is the gecode space associated with the mozart space, `iva` is the object storing the constraint variables, `ia` contains the coefficients, `irt` is the relation type and `c` is the constant.

If an error occurs during the first three operations an exception is thrown in the virtual machine indicating the problem. X_g and Y_g are created from the indexes stored in `X` and `Y`.

5 Constraint propagation

The preceding sections described how constraint variables and constraint posting are implemented. This section is devoted to the constraint propagation mechanism. For understanding this it is important to first consider a subtle difference between the way constraint propagation is triggered in the old Mozart design and in Gecode.

Constraint propagation in Gecode is triggered *on-demand*. This means that the Gecode programmer has to explicitly call a method (called `status`) on a `Space` object in order to trigger propagation. Thus, adding a propagator to a Gecode space does not imply that the propagator is executed. In contrast, in Mozart propagation is executed eagerly. That is, posting a propagator launches its execution. To support advanced Gecode search features such as *batch recomputation* constraint propagation cannot be eager. Therefore, the new Mozart implementation adopts the *on-demand* propagation approach.

Another important issue is the fact that in Mozart threads and propagators are similar entities. They are similar in the sense that both represent concurrent agents that work on a constraint store. The similarity is reflected by the fact that both threads and propagators inherit from the same class (called `Runnable`) in the virtual machine implementation. Moreover, it is possible for threads to suspend on constraint variables as shown in the following Oz program.

```
proc{CSP Root}
  X :: 1#10
  Y :: 2#8
in
  thread
    {Wait X}
    if X > 3 then Y :< 5 end
  end
  % Post propagators on X
  % Branch on X
end
```

The program illustrates the kind of interaction between threads and propagators that we need to support. The thread is suspended until the value of the variable `X` is determined. Then posting or not constraint `Y :< 5` depends on that value. Propagators and threads can thus synchronize via constraint variables or, in general, logic variables. This means that determining when a space represents a fixed-point depends on both propagation and thread execution. In [4] Schulte proposed a way to detect space stability by counting the runnable agents in a computation space. When this counter is equal to zero the space is said to be stable (or at a fixed point).

That notion of stability needs to be modified to take into account virtual machine threads and propagators in the new implementation. To do this in the less "intrusive" possible way we chose to perform constraint propagation in Gecode from a Mozart thread. That is, whenever constraint variables exist there exists also an extra artificial thread (created as an instance of `PropagateThread`).

This thread is in charge of executing the propagation of the associated Gecode space. The thread will suspend on an internal variable (called `StatusVar`, already present in the old design) that internally reflects the space stability.

In the old design, any thread that suspends on `StatusVar` requires the space to be stable. The effect of suspending on that variable will thus trigger the execution of all the runnable threads and hence, of propagation. A disadvantage of this approach is that threads and constraints will not interleave as in the old design. However, this conservative modification allows to grasp a clear understanding of its effects and, therefore, to avoid difficult bugs. Finally, to guarantee a fixed-point, the propagation thread will re-suspend again on `StatusVar` if there are runnable threads after its execution. Any other runnable thread can potentially introduce new propagators and therefore requires a new propagation of the Gecode space.

```
void run(VM vm) {
  Space *sp = vm->getCurrentSpace();
  GecodeSpace *gs = sp->getCstSpace();
  ExecStatus st = gs->status(); // gecode propagation
  if(st == ES_FAILED) {
    sp->setFailed();
    return;
  }
  if(sp->runnableThreads() != 0) {
    this->suspendOn(sp->getStatusVar());
  }
}
```

The propagation thread is only required when there is at least one constraint variable. To avoid any overhead when no constraint programming is being used the thread can be created on demand. Figure 2 describes the situations in which a propagation thread exists.

6 Branching

Current Mozart implementation allows the use of built-in and user-defined branching strategies. Both kinds of branchers are useful and therefore are planned in the integration with Gecode. Initially, we will provide support for the most general type of branchers, those written in Oz. Not including Gecode branchers may seem restrictive. In fact, making Gecode branchers available from Mozart poses several design challenges and decisions such as managing the interaction between different kinds of branchers. Consider the following example:

```
proc{CSP Root}
  X := 1#10
  Y := 2#100   in
% some constraints on X and Y here
  {FD.builtinBrancher X}
  {FD.ozBrancher Y}
end
```

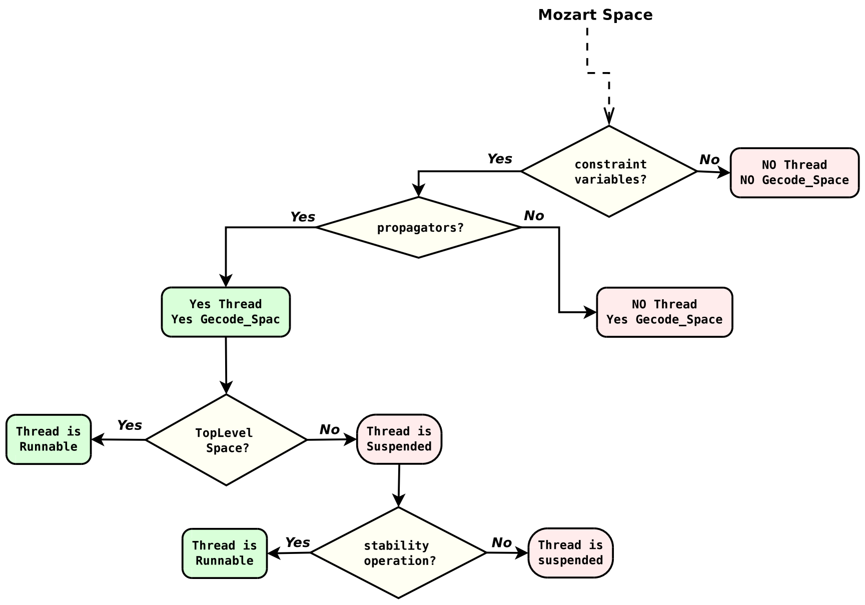


Fig. 2: Lazy creation of Gecode space and propagation thread

Suppose that `{FD.builtinBrancher X}` posts a brancher provided by Gecode. Furthermore, suppose the brancher posted by `{FD.ozBrancher Y}` is written in Oz. At some point during the solving process the second brancher may become active and its execution will thus run Oz code from Gecode. We are still investigating how do to that in language and implementation-safe ways.

Moreover, to support batch recomputation[2] enabled search engines, every computation space would have to maintain an ordered collection of branchers. This is contrary to the Mozart design in which only one brancher[5] is active at a time. Having only one active brancher is assumed for the stability check of Mozart spaces.

For the reasons explained above and with the goal of providing an implementation capable of solving Constraint Satisfaction Problems specified in Oz, we decided to support only one kind of branchers. Gecode built-in branchers will be integrated in further versions of Mozart. Branchers written in Oz rely on two assumptions:

– They are specified by means of the `choice` construction, and
– there is only one that is active at any given time.

As mentioned before, that has a negative impact on batch-recomputation search engines. A more concrete reason is that the `choice` statement waits until

[5] Or distributor in the Mozart terminology.

the space in which it is executed becomes stable, i.e., until propagation of the space has reached a fix point. The blocking semantics of `choice` does not allow other statements following it to be executed. For example, in the following code two branchers (or distributors) are created but the blocking semantics of `choice` ensures that only one is active.

```
proc{CSP Root}
  X =: 1#10
  Y =: 2#100
in
  C = {FD.reflect.min X}
  choice X =: C [] X != C end
  D = {FD.reflect.med Y}
  choice Y <=: D [] Y >: D end
end
```

The second brancher will only be active after the value of X has been determined by the first one. A solution to the problem of defining batch- recomputation friendly branchers needs to be found. We have already devised some ways but they will not be included in the first release of the Mozart-Gecode integration.

7 Search engines

Gecode offers a set of very useful search engines for Constraint Satisfaction Problems. These engines will not be available in the first version of the integration. The reasons are mostly the same as for the branchers. From the design point of view, every mozart space has an associated Gecode space where constraint propagation is performed. Gecode search engines work on raw Gecode spaces that do not know about the existence of the Mozart VM. As the search engine is supposed to create new computation spaces the garbage collection of unused spaces becomes a non-trivial problem. Moreover, it is still not clear how these new spaces are to be associated to Mozart spaces.

Using a Gecode search engine will require one Mozart space and possibly a large amount of Gecode spaces without their associated Mozart space. Among these, only those spaces representing solutions will be accessible from Mozart. The ones that are not required from the VM point of view need to be garbage collected. We are still evaluating different design decisions to solve handle this.

The most straightforward integration of search engines is to define them in Oz. That implies some code duplication because there will be implementations for the search engines that are already present in Gecode. However, the search engines will use the `Space` module operations and hence, garbage collection will not be an issue. These operations are:

`Space.ask`: Runs propagation of the Gecode space until reaching a fixed point. This operation has a direct counterpart in Gecode: `Space::status`.

`Space.clone`: Creates a copy of a computation space. In this case, copies of both the Mozart space and the Gecode space are created. Gecode has an operation in the `Space` module to do that.

`Space.commit`: Decides to use one of the alternatives present in a computation space. This will produce the addition of new constraints. A commit operation is also present in Gecode.

A fourth operation of the space module is `V = {Space.merge S}`. It is used to merge the computation space S with the space the operation it is called from. As a side effect, it returns the root variable of S containing the variables of the CSP that the user is interested in. Internally, the operation merges the variables and constraints of the two spaces.

This operation cannot be supported in the integration design proposed in this work. The reason is that there is no merge equivalent counterpart in Gecode. It is not possible in Gecode to merge the constraints and the propagators of two spaces. For the purpose of solving CSPs in which information can only be derived by propagation and not by the execution of threads we provide an approximation to `Space.merge` called `Space.dataMerge`.

During the execution of the instruction `V = {Space.dataMerge S}` we take into account two Mozart spaces with their respective Gecode spaces. Let us call S_g the Gecode space associated to S, T the space of V and T_g its corresponding Gecode space. Merging the data of S into T amounts to consider the constraint variables of S_g and to add equality constraints with the corresponding variables of T_g. After that, propagation has to be run on T_g to check whether it is failed or not. If the space is failed then the merge behaves in the same way as merging a failed space in Mozart. If it is not failed then the resulting space represents the result of the operation.

Notice that there is no merging of propagators. Conceptually the propagators in S_g do not exist anymore in T_g. However the equality constraints that were added represent their impact in the data of both spaces. If all the information in S_g can be derived by constraint propagation then this is enough. The case of threads generating new information needs to be studied in more detail and will be part of a forthcoming release.

Having the search engines defined in Mozart greatly facilitates garbage collection. A Gecode space is garbage-collected when its associated Mozart space is collected. That happens when, for example, the reference to the space becomes out of scope. Another advantage of this approach is that reasoning about the garbage collector can rely on the invariant of having one Mozart space per Gecode space.

Depth first search

The fourth space operations defined above can be used to program search engines in an straightforward way. For example, a depth first search engine takes an initial space S containing the CSP to be solved. The search tree is generated by running propagation in S with the Space.ask primitive. After knowing the possible alternatives in which S can evolve we use Space.commit and Space.clone to examine all of them recursively. When a solution is found, Space.dataMerge is used to get the value of the corresponding root variable.

8 Examples

In this section we give Oz implementations for two well-known puzzles. Our goal is to present the differences, which we consider minimal, between the previous FD Mozart versions and this one.

```
proc {Most Root}
   S E N D M O T Y
in
   Root = sol(s:S e:E n:N d:D m:M o:O t:T y:Y)
   {FD.dom 0#9 Root}
   {FD.distinct post(Root)}
   {FD.linear post([1] [S] '\\=:' 0)}
   {FD.linear post([1] [M] '\\=:' 0)}
   {FD.linear post([1000 100 10 1 1000 100 10 1 ~10000 ~1000 ~100 ~10 ~1]
                   [S E N D M O S T M O N E Y] '=:' 0)}
   {FD.distribute Root size_max val_min}
end
proc {Obj O N}
   {FD.linear post([10000 1000 100 10 1 ~10000 ~1000 ~100 ~10 ~1]
                   [N.m N.o N.n N.e N.y O.m O.o O.n O.e O.y] '>:' 0)}
end
{Show {SearchBest Most Obj}}
```

Fig. 3: SEND+MOST=MONEY puzzle

Send+Most=Money Recall the specification of the puzzle SEND+MOST=MONEY from *Modeling and Programming with Gecode.*

> The Send Most Money Problem consists in finding distinct digits for the letters S, E, N, D, M, O, T, and Y such that the well-formed equation (no leading zeros) SEND + MOST = MONEY holds and that MONEY is maximal.

The script can be simply coded into few lines. Figure 3 presents almost same code than that of previous Mozart versions, changed only by the constraint posting syntax and the random strategy for variable selection as its distribution strategy.

N-Queens Consider now another well-known problem. Place N queens on an $N \times N$ chess board such that no two queens attack each other. The parameter of the problem is N.

```
fun {Queens N}
  proc {$ Root}
    L1N = {MakeTuple c N}
    LM1N = {MakeTuple c N}
  in
    {FD.tuple queens N 1#N Row}
    {For 1 N 1
      proc {$ I}
        L1N.I = I
        LM1N.I = ~I
      end}
    {FD.distinct post(Row)}
    {FD.distinct post(LM1N Row)}
    {FD.distinct post(L1N Row)}
    {FD.distribute Row size_min val_min}
  end
end
{Show {SearchAll {Queens 4}}}
```

Fig. 4: Queens puzzle

9 Conclusions

We acknowledge our integration is still ongoing and that optimizations have been left behind. Nevertheless, so far the integration has achieved the proposed goals in a transparent manner. Simply put, the Mozart user only need to adapt to the new syntax for constraint posting and learn which are the new available distribution strategies described in this report. Although the development is not finished yet, forthcoming abstractions in the constraint subsystem will preserve the classic Mozart philosophy, i.e., the declarative nature of the language.

9.1 Future work

To conclude, we would like to present a short overview of our goals for the next stable release. We consider these to be the ones with highest priority given the advantages they will provide.

- Support for sets, booleans and floats domains.
- Space primitives waitStable and choose.

– Support of the Gecode Interactive Search Tool (GIST).
– Support for solving optimization CSPs with optimization functions written in Mozart. This may require Gecode to Mozart space references, extensions to cloning (cloning in Gecode should also clone in Mozart), and garbage collection support (cloned space in Mozart has only reference from Gecode).
– Batch recomputation support as explained before.
– Implementation of space merge as explained before.

10 Acknowledgments

We would like to give special thanks to Gustavo Gomez from Pontificia Universidad Javeriana-Cali, Carlos A. Martínez and Julián Camargo from Universidad del Valle and Sébastien Doeraene from Université Catholique de Louvain for their comments and contribution to this project. We also thank the anonymous reviewers for their comments and suggestions.

References

1. Gecode generic constraint development environment. `http://www.gecode.org`. Accessed: 2013-07-02.
2. Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In *In Proceedings Of The Seventh International Conference On Principles And Practice Of Constraint Programming (cp'01), Lecture Notes In Computer Science*, pages 240–255. Springer-Verlag, 2001.
3. Tobias Müller. *Constraint Propagation in Mozart.* Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.
4. Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 2002.
5. Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto,Canada, September 2004. Springer-Verlag.
6. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008.