Implementing a Distributed Shortest Path Propagator with Message Passing

Luis Quesada, Stefano Gualandi, and Peter Van Roy

Université Catholique de Louvain
Place Sainte Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
{luque, stegua, pvr}@info.ucl.ac.be

Abstract. In this article, we present an implementation of a distributed shortest path propagator. Given a graph and a goal node, this propagator maintains a finite domain variable for every node. The variable's lower bound is the minimal cost of reaching the goal from that node. The graph on which the propagator is based can be modified by removing edges, increasing the cost of edges, or by replacing edges by graphs. The propagator has been implemented using a message passing approach on top of the multi-paradigm programming language Oz [2]. One of the advantages of using a message passing approach is that distributing the propagator comes for free. Different shortest path propagators running on different machines may be working together on the same graph.

1 Introduction

In this article, we present the implementation of a distributed shortest path propagator. Given a graph and a goal node, this propagator offers the following services:

- It maintains, for every node, a Finite Domain (FD) variable. The lower bound of the variable is the minimal cost of reaching the goal from that node. The propagator maintains these lower bounds while the graph is dynamically modified.

- It allows the incremental definition of the graph on which the propagator is based. The user may start by providing an abstract graph (i.e., a graph whose edges are virtual) and then proceed by replacing each edge by its corresponding graph. The user can execute the propagator in a distributed way, since he can choose to launch the propagator associated with a virtual edge on a different machine.

The graph on which the propagator is based can be modified either by increasing the cost of an edge or by deleting an edge[1]. It is important to emphasize that we only consider monotonic changes in the graph (i.e., changes that lead to further

[1] The reader may also think of deleting an edge as increasing its cost to ∞. However, mostly for optimization reasons, we prefer to keep the two operations separate.

constrain the FD variables involved).
In order to maintain the minimal costs (and thus the FD variables) we follow the asynchronous dynamic algorithm described by [9], which is based on the principle of optimality.
Let us represent the shortest distance from node i to the goal node as $h^*(i)$. The shortest distance via a neighboring node j is given by $f_i^*(j) = k(i,j) + h^*(j)$, where $k(i,j)$ is the cost of the link between i and j. If node i is not the goal node, the path to the goal node must visit one of the neighboring nodes. Therefore, $h^*(i) = min_j f_i^*(j)$ holds.
If h^* is given for each node, the optimal path can be obtained by repeating the following procedure. For each neighboring node j of the current node i, compute $f_i^*(j) = k(i,j) + h^*(j)$. Then move to the j that gives $min_j f_i^*(j)$.
Asynchronous dynamic programming computes h^* by repeating the local computations at each node. Let us assume the following situation:

1. For each node i, there exists a process corresponding to i.
2. Each process records $h(i)$, which is the estimated value of $h^*(i)$. We initialize $h(i)$ to ∞.
3. For the goal node g, $h(g)$ is 0.
4. Each process can refer to the h values of neighboring nodes.

Each process updates $h(i)$ by the following procedure. For each neighboring node j, compute $f_i(j) = k(i,j) + h(j)$, where $h(j)$ is the current estimated distance from j to the goal node, and $k(i,j)$ is the cost of the link from i to j. Then, update $h(i)$ as follows: $h(i) \leftarrow min_j f_i(j)$. The execution order of the processes is arbitrary.
There are several ways of implementing the algorithm sketched above. We implement it by considering the set of processes as a multi-agent system where agents interchange synchronous and asynchronous messages and their transition state functions rely on data flow and constraint programming primitives.
The organization of the paper is as follows. In Section 2, we introduce the Oz language and explain how message-passing concurrency can be modeled in it. In Section 3, we show how to use the propagator and explain the implementation of it by showing its message passing diagram and its corresponding state transition system. We conclude in Section 4.

2 Message-passing concurrency in Oz[2]

2.1 The Oz language and its Execution Model (Declarative Subset)

The declarative part of the Oz execution model consists of a store and a set of dataflow threads that reference logic variables in the store (see Figure 1). Threads contain statement sequences S_i and communicate through shared references. A thread is a *dataflow* thread if it only executes its next statement when all the values the statement needs are available. Data availability is implemented using

[2] This section is a summary of section 2 of [7], and Chap. 5 of [6]

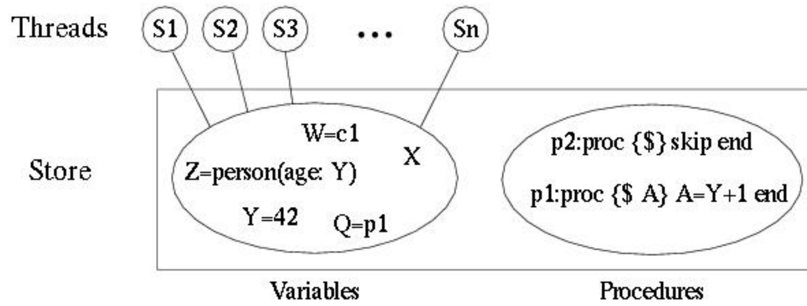

Fig. 1. The Oz execution model (Declarative Subset).

logic variables. If the statement needs a value that is not yet available, then the thread automatically blocks until the value is available. There is also a fairness condition: if all values are available then the thread will eventually execute its next statement.
The shared store is not physical memory; rather it is an abstract store that only allows legal operations for the entities involved, i.e., there is no direct way to inspect their internal representations. The store consists of two compartments, namely logic variables (with optional bindings) and procedures (named lexically scoped closures). Variables can reference the names of procedures. The external references of threads and procedures are variables. When a variable is bound, it disappears, i.e., all threads that reference it will automatically reference the binding instead. Variables can be bound to any entity, including other variables. The variable and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed. Because of monotonicity, a thread that is not blocked is guaranteed to stay not blocked until it executes its next statement.
All Oz execution can be defined in terms of a kernel language whose semantics are outlined in [1], [8] and [6]. We will just describe the declarative part of it.

$S ::= S\ S$	Sequence
$\mid X = f(l_1 : Y_1 \ldots l_n : Y_n) \mid$	Value
$\mid X =$<number> $\mid X =$<atom>	
$\mid$ **local** $X_1 \ldots X_n$ **in** S **end** $\mid X = Y$	Variable
$\mid$ **proc** {X Y$_1 \ldots Y_n$}S **end** $\mid$ {X Y$_1 \ldots Y_n$}	Procedure
$\mid$ **if** X **then** S **else** S **end**	Conditional
$\mid$ **thread** S **end**	Thread

Table 1. The Oz declarative kernel language.

Table 1 defines the abstract syntax of a statement S in the (declarative part of the) Oz kernel language. Statement sequences are reduced sequentially inside a thread. All variables are logic variables, declared in an explicit scope defined by the local statement. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables. Procedures are defined at run-time with the **proc** statement and referred to by a variable. Procedure applications block until the first argument references a procedure name. The **if** statement defines a conditional that blocks until its condition is **true** or **false** in the variable store. Threads are created explicitly with the **thread** statement.

In the following section, we are going to be using a bit of syntactic sugar to make programs easier to read. We will do so by considering that:

- **proc** {P V1 V2 ... Vn} <Decl> **in** <Stmt> **end** is equivalent to **proc** {P V1 V2 ... Vn} **local** <Decl> **in** <Stmt> **end** **end**, where <Decl> is a declaration (i.e., a statement declaring a variable) and <Stmt> is any statement.
- **fun** {F V1 V2 ... Vn} <Stmt> <Exp> **end** is equivalent to **proc** {F V1 V2 ... Vn O} <Decl> **in** <Stmt> O=<Exp> **end**, where <Exp> is an expression representing a value.
- **fun** {F V1 V2 ... Vn} <Decl> **in** <Exp> **end** is equivalent to **fun** {F V1 V2 ... Vn} **local** <Decl> **in** <Exp> **end** **end**.

Procedures are values in Oz. This means that a variable may be bound to a procedure. In particular, we have that **proc** {X V1...Vn}... **end** is equivalent to X=**proc** {$ V1...Vn}... **end**.

2.2 The message-passing concurrent model

The message-passing concurrent model extends the declarative concurrent model by adding ports. Ports are a kind of communication channel. Ports are no longer declarative since they allow observable nondeterminism: many threads can send a message to a port and their order is not determined. However, the part of the computation that does not use ports is still declarative.

Ports. A port is an Abstract Data Type (ADT) that has two operations:

{NewPort S P}: create a new port P associated with stream S.
{Send P X}: append X to the stream corresponding to the entry point P. Successive sends from the same thread appear on the stream in the same order in which they were executed. This property implies that a port is an asynchronous FIFO communication channel.

For example:

```
local S P in
   {NewPort S P}
```

```
   {Send P a}
   {Send P b}
   {Browse S}
end
```

This displays the stream `a|b|_`. Doing more sends will extend the stream. By asynchronous we mean that a thread that sends a message does not wait for reply; it immediately continues.

Port objects. A port object is a thread reading messages from port streams. This allows two things. First, many-to-one communication is possible: many threads can reference a given port object and send to it independently. Second, port objects can be embedded inside data structures (including messages).
Here is an example of a port object with one port that displays all the messages it receives:

```
local S P in
   {NewPort S P}
   thread {ForAll S proc {$ M} {Browse M} end} end
end
```

In this example, `ForAll` is a procedure that, given a list `L` and a procedure `P`, applies `P` to each element of `L`. Doing `{Send P hello}` will eventually display `hello`.

The NewPortObject abstraction. We can define an abstraction to make it easier to program with port objects. Let us define an abstraction for the case that the port object has just one port. To define the port object, we give the initial state `Init` and the state transition function `Fun`, which is of type $State \times Msg \rightarrow State$.

```
proc {NewPortObject Fun Init ?P}
   proc {MsgLoop S1 State}
      case S1 of Msg|S2 then
         {MsgLoop S2 {Fun Msg State}}
      [] nil then skip end
   end
   Sin
in
   thread {MsgLoop Sin Init} end
   {NewPort Sin P}
end
```

3 The Shortest Path Propagator

3.1 Interface of the Propagator

As shown in Figure 2, we need to specify the graph on which the propagator is based and the node in the graph that is the goal. The representation of the

graph is an adjacency list. The function `Create_Propagator` returns a record representing the interface of the propagator.
The propagator has the following interface:

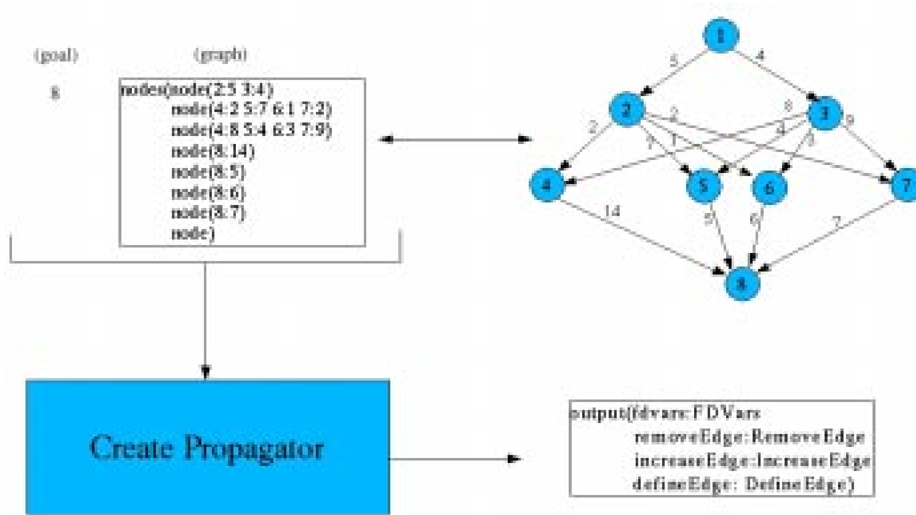

Fig. 2. Description of the propagator.

`fdVars` exports a tuple `T` of FD variables. `T.i` is the FD variable associated with node `i`. This FD variable corresponds to the cost of going from that node to the goal.

`removeEdge` exports a 1-argument procedure:
proc `{$ Edge}` ... **end**. The parameter `Edge` is the edge to be removed.

`increaseEdge` exports a 2-argument procedure:
proc `{$ Edge NewCost}` ... **end**. The parameter `Edge` is the edge whose cost is to be increased to `NewCost`[3].

`defineEdge` exports a 5-argument function:
fun `{$ Edge Graph Source Destination Host}` ... **end**. `Edge` is the edge to be replaced. `Graph` is the graph by which the edge is to be replaced. `Source` is the node in `Graph` with which the origin of `Edge` is to be associated. `Destination` is the node in `Graph` with which the destination of `Edge` is to be associated. `Host` is the url address of the host on which the propagator of `Graph` is going to be executed. If `Host` is nil, the propagator will be executed on the same machine.

In Oz, we use the following notation to represent FD variables: VariableName-{LowerBound#UpperBound}. In the implementation, the upper bounds of the

[3] The implementation assumes that the new cost is always greater than the current cost.

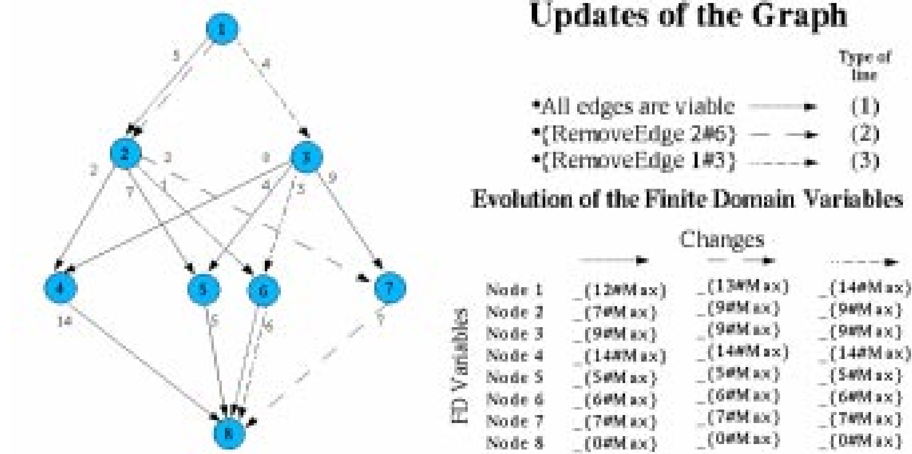

Fig. 3. Updating the FD variables of each node.

propagator's FD variables are set to a constant `Max`. This constant could be the sum of the edges' costs. As shown in Figure 3, after the creation of the propagator, the lower bound of the FD variable of each node is set to the minimal cost of reaching the destination. Internally, the propagator always keeps the shortest path to the destination for every node. In Figure 3, we show how the shortest path from node 1 to the goal node (node 8) is updated after removing edges `2#6` and `1#3`. In this Figure, you can also observe how the FD variables are updated with respect to the changes in the graph[4].

3.2 Implementing the propagator

The propagator is implemented as a set of port objects interchanging asynchronous and synchronous messages. The implementation makes use of the NewPortObject abstraction described in the previous section. In this section, we will describe the messages that our objects interchange, the attributes that these port objects have and their corresponding state transition functions. Even though Figure 4 shows three types of concurrent processes, only nodes are modeled as port objects since neither the environment nor the monitors receive messages. In the following, we will focus on the implementation of the nodes.

Attributes of the Nodes. As shown in Figure 4 each node has the following attributes:

`OutNodes`: the tuple of outgoing nodes[5].

[4] An edge is represented as a tuple of two elements. The edge `Ind1#Ind2` has `Ind1` as origin and `Ind2` as destination.

[5] `Y` is an outgoing/incoming node of `X` if `Y` is the destination/origin of one of `X`'s outgoing/incoming edges.

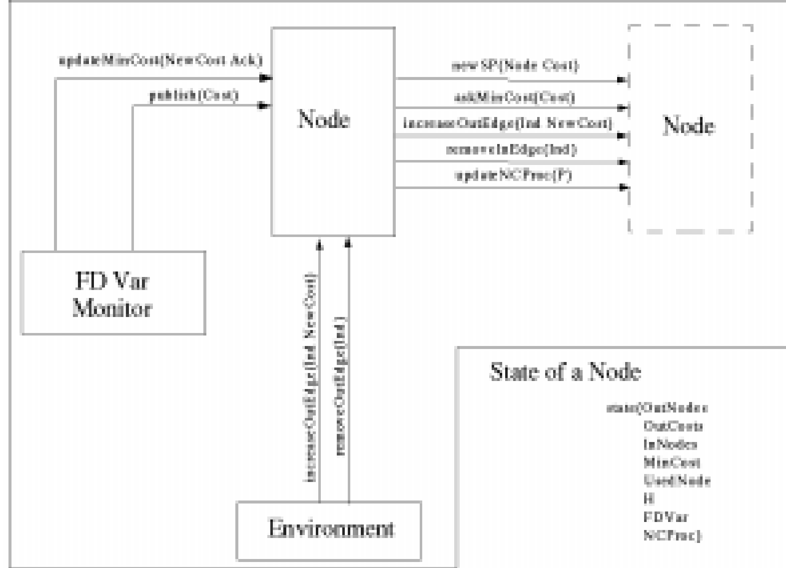

Fig. 4. Message Diagram and State of a Node.

`OutCost`: the tuple of costs of reaching the outgoing nodes.
`InNodes`: the tuple of incoming nodes.
`MinCost`: the minimum cost of reaching the destination.
`UsedNode`: the outgoing node that is being used to reach the destination.
`H`: the priority queue that keeps the outgoing nodes that are not being used. Each one of these nodes is associated with a key that represents the cost of going to the destination through that node.The priority queue is implemented with a heap.
`FDVar`: the FD variable maintains the cost of reaching the destination. One of the invariants of our system is that the lower bound of the FD variable is equal to the minimal cost of reaching the destination.
`NCProc`: the procedure to be executed whenever `MinCost` is updated. As we will see in Section 3.4, the implementation of virtual edges makes use of this attribute.

Message Diagram. Figure 4 shows the Message Diagram of the Shortest Path propagator. There are three kinds of entities: (*i*) nodes, (*ii*) FD variable monitors and (*iii*) environment. A node may receive messages from another node, from its FD variable monitor and from the environment. By environment, we mean all those entities independent to the propagator that may interact with it.

The messages that the environment may send to a particular node are:

`increaseOutEdge(Ind NewCost)`. It increases the cost of edge **`self`**`#Ind` to `NewCost` and updates the state of the node accordinly.

`removeOutEdge(Ind)`. It removes the edge **`self`**`#Ind` and updates the state of the node accordinly.

A set of propagators in a Constraint Satisfaction Problem communicate with each other through shared FD variables. The FD variables associated with each node may be updated by other propagators working concurrently with the Shortest Path Propagator. So, as the changes in the FD variables depend not only on the Shortest Path Propagator, a concurrent process (namely the FD Variable Monitor) is responsible for detecting those changes and updating the minimal cost of the corresponding node. The messages that a FD variable monitor may send to its node are:

`updateMinCost(NewCost Ack)`. It updates the minimal cost of the node and binds `Ack` once the cost has been updated. The `Ack` parameter is for the sender to wait until the execution of the message has finished. As we are using port objects, messages are asynchronous by default, so this mechanism is a way of modeling synchronous messages.

`publish(Cost)`. It communicates the new cost to the corresponding incoming nodes.

The messages that a node may send to another node are:

`newSP(Node Cost)`. It updates the state of the node according to the fact that node `Node` has a new minimal cost `Cost`.

`askMinCost(Cost)`. It binds `Cost` to the current minimal cost of the node.

`increaseOutEdge(Ind NewCost)`. It increases the cost of edge **`self`**`#Ind` to `NewCost` and updates the state of the node accordingly.

`removeInEdge(Ind)`. It removes edge **`self`**`#Ind` and updates the state of the node accordingly.

`updateNCProc(P)`. It updates `NCProc`.

3.3 State Transition

As we are using port objects, our algorithm is reduced to specifying how the state of our port objects evolve when receiving a particular message. We will consider two cases:

`removeOutEdge(Ind)`. As shown in Figure 5, there are two possibilities for the node when receiving this message depending on whether the edge to be removed is the one being used. If it is the used edge, another outgoing node is chosen from the priority queue, the FD variable and the minimum cost are updated and the procedure `NCProc` is executed. If it is not the edge used, the information of the edge is simply removed.

`newSP(Node Cost)`. As shown in Figure 6, the state transition for this message involves more cases. If `Node` is not the used node, the heap is updated with the new key for `Node`. If `Node` is the one used, there are two cases. One case is when **`self`** only has one outgoing node. If this is the case, there is no option

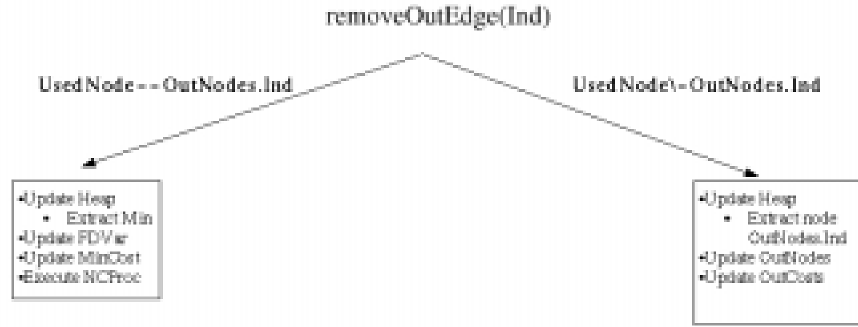

Fig. 5. State Transition for the message removeOutEdge.

but to keep using the same node. So, the FD variable and the minimal cost are updated according to `Cost` and the `NCProc` is executed. The other case is when **`self`** has more than one outgoing node. In this case there are two sub-cases. One sub-case is when the best option (to go to the destination) offered by the nodes in the heap is worse than the new option offered by the used node. In this sub-case, we simply update the FD variable and the minimal cost according to `Cost` and execute `NCProc`. Otherwise, we have to update the heap by extracting the node offering the best cost and inserting the current used node. We also have to update the FD variable, the minimal cost, the used node, and execute `NCProc`.

3.4 Dealing with virtual edges

The shortest path propagator allows the user to define the graph on which the propagator is based incrementally. It does so by letting the user associate graphs to edges. Thanks to the approach chosen to implement the propagator, the implementation of this facility is straightforward :

```
fun {DefineEdge Edge Graph Source Destination Host}
   if Host == nil then
      Ind1#Ind2=Edge
      SPP={Create_Propagator Graph Destination}
      proc {NCProc}
         thread {IncreaseEdge Ind1#Ind2 {SPP.askMinCost Source}} end
      end
   in
      {SPP.updateNCProc Source NCProc}
      SPP
   else ... end
end
```

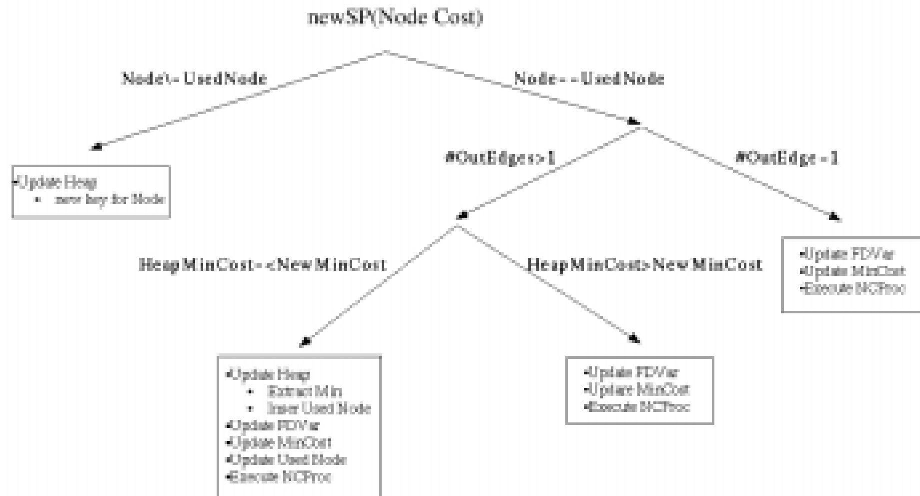

Fig. 6. State Transition for the message newSP.

An independent shortest path propagator is created for `Graph`. The port object associated with `Source` is set so that it sends an `increaseEdge` message to the origin of `Edge` whenever the corresponding `MinCost` of `Source` is updated. Here, we only show how to implement the case where all the concurrent processes are created on the same machine. Readers interested in seeing how this approach could be extended to manage the case where the processes are created on different machines may read Chap. 11 of [6].

4 Conclusion

We have presented the Shortest Path Propagator. Given a graph and a destination node, this propagator maintains, for every node, a finite domain variable whose lower bound is the minimal cost of reaching the destination from that node. Even though this is a problem already investigated (see for instance [5][6] and [9]), the value of our work lies in the fact that the algorithm is implemented using a message-passing approach on top of data flow and constraint programming primitives. The use of this sophisticated approach allows, for instance, the

[6] The propagator can also be used as an incremental algorithm for the single source shortest path problem. However, our incremental algorithm (besides only supporting monotonic changes in the graph) is not as efficient as the algorithm presented in [5]. We can only update MinCost monotonically (i.e., the value to which MinCost is updated must always be greater than the current one) because this variable is associated with the lower bound of a FD variable. [5] can perform better by allowing MinCost to be updated non monotonically (e.g., by temporarily setting affected nodes' cost to be too high) even though the changes to the graph are monotonic.

easy extension of the propagator to deal with the incremental definition of the graph on which the propagator is based. Another extension that comes for free is the promotion of the propagator to a distributed status. Different shortest path propagators running on different machines may be working together on the same graph.
We identify, at least, two scenarios where the presented propagator may be of great utility. One is for solving TSP derived problems using a hierarchical approach. [4], for instance, considers cases where the graph on which the problem is based is explored by demand. The other scenario has to do with the implementation of propagators for the same kind of problem which duty is to prune non-viable edges. If the cost of the nodes represent time, those nodes may be associated with time windows (i.e., the node can only be visited within some time periods). [3] suggests, for instance, the use of shortest path propagators for inferring nodes that have to be visited before others (due to the presence of time windows), thus avoiding failures during the search phase.

5 Acknowledgments

Special thanks are due to Kevin Glynn, who helped us to improve the quality of our paper with his valuable comments. We would also like to thank the Mozart Research group (at UCL) for their comments on the design of the shortest path propagator.

References

1. S. Haridi and N. Franzen. *Tutorial of Oz.* December 1999. Available at `http://www.mozart-oz.org/`.
2. Mozart Consortium. *The Mozart Programming System Version 1.2.5.* December 2002. Available at `http://www.mozart-oz.org/`.
3. G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows. *Transportation Science*, 32:12–29, 1998.
4. L. Quesada and P. Van Roy. A concurrent constraint programming approach for trajectory determination of autonomous vehicles. In *CP 2002 Proceedings*, 2002.
5. G. Ramalingam and T.W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
6. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* 2003. To be published by MIT Press. Expected publishing date 2004.
7. P. Van Roy, S. Haridi, P. Brand, M. Mehl, R. Scheidhauerand, and G. Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.
8. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.
9. G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, Cambridge, MA, 1999.