A Coordination-Based Framework for Parallel Constraint Solving

Peter Zoeteweij

Centrum voor Wiskunde en Informatica
P.O. Box 94079, NL-1090 GB AMSTERDAM
P.Zoeteweij@cwi.nl

Abstract. This work in progress paper gives an overview of DICE (Distributed Constraint Environment), a framework for the configuration of parallel constraint solvers. DICE is implemented using the Manifold coordination language, and provides coordination services to software components in four categories: (1) variable domain types, (2) (incomplete) solvers, (3) splitting methods that build search trees, and (4) search strategies that traverse these search trees. In addition to the coordination services and the algorithms that they implement, the paper describes the configuration of solvers both from a constraint programming and a software engineering point of view.

1 Introduction

The applicability of the Ideal Worker Ideal Manager (IWIM) coordination model in the area of distributed constraint solving has been studied in [2, 8]. This work has materialized in DICE (Distributed Constraint Environment), a software framework built in the Manifold coordination language, where constraint solvers are constructed from software components in four categories:

1. domain types for the variables of a constraint satisfaction problem (CSP),
2. constraint solvers that reduce a CSP to an equivalent but simpler problem,
3. methods to split the domains of variables in order to build a search tree, and
4. search strategies that determine the order in which to traverse the nodes of this search tree.

Each component instance can run in a separate process. The coordination services that the DICE framework provides to the components implement

- a distributed constraint propagation algorithm,
- a distributed termination detection algorithm, and
- facilities that allow a split method component and a search strategy component to coordinate the network of processes to perform search.

The constraint propagation algorithm applies (incomplete) solvers until none of these can reduce the problem any further. In general, this will not lead to a solution to the CSP, and propagation must be interleaved with splitting the domain of a variable in order to systematically search for solutions. Termination detection is needed because usually, we don't want to split the domain of a variable until constraint propagation has finished.

The aim of this paper is to give an overview of DICE. In Sect. 2 we describe the model of constraint solving that it is based on, and provide pointers to more information about coordination programming, the IWIM model, and Manifold. Also here we recall the algorithms and facilities implemented by the coordination services. In Sect. 3 and 4 we describe solver configuration from constraint programming and software engineering perspectives, respectively, and Sect. 5 is an overview of a set of components that is available for the construction of solvers. In Sect. 6 we discuss related work, and our plans for future work on DICE.

2 Preliminaries

2.1 Constraint Solving

The model of constraint solving supported by DICE is characterized by

- interleaving of constraint propagation and splitting,
- constraint propagation by domain reduction, and
- application of domain reduction functions (DRF's) by means of a distributed constraint propagation algorithm.

When propagation finishes in a node of the search tree, one of three situations occurs: if the domains of all variables have been reduced to singletons, the (leaf) node represents a solution. If one or more variable domains have become empty, the node represents a failure. In all other cases, the node is an internal node of the search tree, and the *splitting method* determines how the search tree is expanded.

The work of the splitting method consist of selecting a variable for splitting, and determining the domains for that variable in the resulting sub-CSP's. These two aspects are referred to as *variable selection* and *value selection*, respectively.

The collection of rules for selecting the nodes of the search tree where the constraint propagation algorithm is applied in the course of the solving process is referred to as the *search strategy*.

It is important to note that the model of constraint solving supported by DICE does not involve constraint checking. Violation of a constraint is concluded when the domain reduction functions that enforce the constraint void the domains of one or more variables.

2.2 Coordination Model and Language

Coordination languages offer language support for composing and controlling software architectures made of parallel or distributed components [9]. In the IWIM model of coordination [3], these components are represented by *processes*. In addition to processes, the basic concepts of IWIM are *ports*, *channels* and *events*. A process is a *black box* that exchanges *units* of information with the other processes in its environment through its input ports and output ports, by means of standard I/O primitives analogous to read and write. The interconnections between the ports of processes are made through directed channels. Independent of channels, there is an event mechanism for information exchange in IWIM. Events are broadcast by their sources, yielding event *occurrences*. Processes can tune in to specific event sources, and react to their occurrences.

The IWIM view of a software system is a dynamic ensemble of interconnected processes. A process can be regarded as a worker process or a manager process. The responsibility of a worker process is to perform a (computational) task. The responsibility of a manager is to coordinate the communications among a set of worker processes. For this purpose, manager processes can create worker processes and make channel connections to their ports. A manager process may be considered a worker processes by another manager. At the bottom of this hierarchy there is always a layer of *atomic workers*.

Manifold [3,4] is a coordination language for writing program modules (coordinator processes) to manage complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes that comprise a single application. The conceptual model behind Manifold is based on IWIM. A Manifold application consists of a (potentially very large) number of processes running on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages and some of them may not know anything about Manifold, nor the fact that they are cooperating with other processes through Manifold in a concurrent application.

2.3 A Distributed Constraint Propagation Algorithm

The purpose of a constraint propagation algorithm is to apply atomic reduction steps. For the model of constraint solving supported by DICE, these atomic reduction steps are domain reduction functions. Many constraint propagation algorithms maintain a set of atomic reduction steps that still need to be applied, and keep applying reduction steps until this set becomes empty. After each reduction step, the algorithm considers the changes that have been made, and reduction steps that depend on these changes are added to the set.

In contrast to such inherently sequential algorithms, the propagation engine of DICE implements the coordination-based chaotic iteration algorithm of [8]. In this algorithm, each CSP variable is represented by a process, that holds the domain of the variable. Also each domain reduction function is represented by a process that receives input from the processes corresponding to the CSP variables that the function applies to. The DRF processes have a buffer associated with each input port, that stores the variable domain last seen on that input port. These buffers are initialized by having a Variable process send its domain each time a connection to a DRF process is made.

Figure 1(a) shows an example of a process network of the distributed constraint propagation algorithm. Variable processes send *reduction requests* to DRF processes. Such reduction requests contain the domain of the CSP variable. The DRF process uses this domain to update the buffer associated with the input port that delivers the reduction request. Then it applies the domain reduction function to the domains in the buffers[1]. This yields new domains for the output variables of the domain reduction function. These domains are dispatched through the output ports of the DRF process to the corresponding Variable processes as *update commands*.

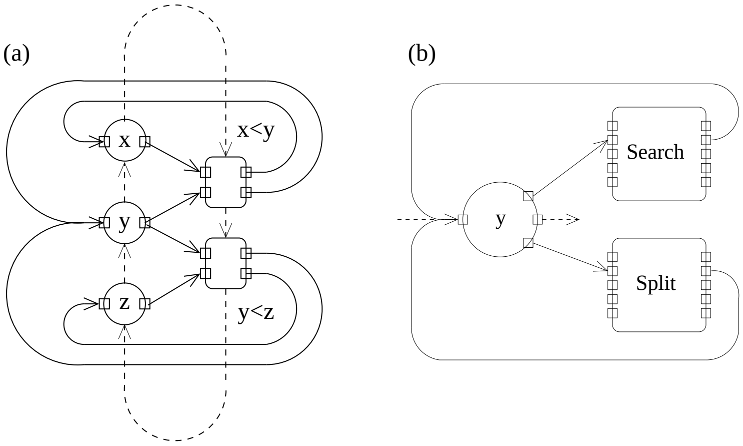

Fig. 1. (a) An example process network of the distributed constraint propagation algorithm, (b) all Variables are connected to the Split and Search processes

Upon receiving an update command, a Variable process computes the intersection of the domain held in the update command and the domain of the CSP variable held in its internal store. If this intersection is a proper subset of the domain of the CSP variable in the internal store, the store is updated with the intersection, and the new domain of the CSP variable is dispatched through the output port of the Variable process as a reduction request. The reduction request is broadcast to all DRF processes that connect to this output port. If the intersection does not reduce the domain of the CSP variable, the update command has no effect.

2.4 Termination Detection

Although this is not strictly necessary, usually we do not want to consider expanding the search tree by splitting the domain of a variable before constraint

[1] Actually, application of the domain reduction function is postponed until no more reduction requests are immediately available on the input ports. Such details are omitted from this presentation.

propagation has finished. Therefore, we need to know when the propagation algorithm terminates. With a sequential algorithm, this is easy: it terminates when the set of atomic reduction steps that still need to be applied becomes empty.

In the case of the distributed algorithm of the previous section, the conditions for concluding that constraint propagation has finished are more difficult to verify. The algorithm terminates when:

1. all Variable and DRF processes are idle, and
2. there are no pending update commands or reduction requests in the channels.

DICE employs the algorithm described in [5] to detect these conditions. For the purpose of this algorithm, the processes of the constraint propagation algorithm are connected in a ring network. The dashed lines in Fig. 1(a) show the extra channels for termination detection. All nodes maintain a local counter of the number of messages in the network. The ring network is used for circulating a token. When it returns to the node that created it, the token has accumulated the local counters of all nodes. Termination can be concluded only if this sum equals 0. Together with a black/white coloring of nodes and token the algorithm ensures correctness in case of buffered channels. This corresponds to the Manifold communication model.

2.5 Distributed Splitting

DICE employs a scheme similar to that of [2], where a network of processes of the constraint propagation algorithm is performing work in several nodes of the search tree simultaneously. All administration inside the Variable and DRF processes for the purpose of the propagation and termination detection algorithms is per node of the search tree:

$\mathit{Variable}$::	DRF::
v: $\mathit{World} \stackrel{m}{\rightarrow} \mathit{Domain}$	I: ARRAY $[1..n]$ OF $\mathit{World} \stackrel{m}{\rightarrow} \mathit{Domain}$
color: $\mathit{World} \stackrel{m}{\rightarrow} \{\text{black}, \text{white}\}$	color: $\mathit{World} \stackrel{m}{\rightarrow} \{\text{black}, \text{white}\}$
$\mathit{n\_msg}$: $\mathit{World} \stackrel{m}{\rightarrow} \mathbb{N}$	$\mathit{n\_msg}$: $\mathit{World} \stackrel{m}{\rightarrow} \mathbb{N}$

where n is the number of input ports of the DRF process, and *World* is a datatype whose elements serve as identifiers for nodes of the search tree. A partial order is defined on the elements of *World*, stating that an ancestor node is compatible to its descendants, and that a descendant is smaller than its parent. On several occasions, we look for information in the smallest compatible world of a world w. For example, the update commands of the propagation algorithm now consist of a world w, and a domain d. If the world w is not yet known to the Variable process, it intersects d with the domain d' of the CSP variable in the smallest compatible world of w. Only if $d' \cap d \subset d'$, the element $w \rightarrow d' \cap d$ is added to the map v of the Variable process.

The facilities offered by this administration per world are used by two new processes *Split* and *Search*, that implement the splitting method (involving variable selection and value selection) and search strategy, respectively. These processes have connections to all Variable processes (Fig. 1(b)).

The Split process is triggered when propagation finishes in a certain world, and may query Variable processes for their domains in that world. On the basis of this information, the Split process can then decide which variable to split (if any), and construct a new set of *World-Domain* pairs for that variable. The worlds of this set correspond to the sub-CSP's created by the split.

Upon receiving new worlds and corresponding domains from the Split process, a Variable process tells the Search process about these new worlds. This allows the Search process to maintain an administration of worlds where the constraint propagation algorithm still needs to be applied. The Search process coordinates the activities of the propagation network through the search tree, by issuing commands that start propagation in worlds that it knows about. In the current implementation of DICE, the Search process may consider starting propagation in a new world on two occasions: when propagation finishes in a certain world, and when a Variable process notifies that it has received new worlds from the Split process.

3 Solver Configuration

From a user's point of view, solvers are configured in a small language. The following example shows a solver configuration for the SEND + MORE = MONEY puzzle.

```
VARIABLE s IS DiscreteDomain "1..9";
VARIABLE e IS DiscreteDomain "0..9";
...
VARIABLE y IS DiscreteDomain "0..9";
DRF IntegerArithmetic "s != e";
DRF IntegerArithmetic "s != n";
...
DRF IntegerArithmetic "r != y";
DRF IntegerArithmetic
   "            1000*s + 100*e + 10*n + d "
   "          + 1000*m + 100*o + 10*r + e "
   " = 10000*m + 1000*o + 100*n + 10*e + y ";
```

This example demonstrates the `VARIABLE` and `DRF` statements, used to set up the process network described in Sect. 2.3. Figure 2 shows the relevant part of the solver configuration language syntax.

The functionality of the Variable, DRF, Search and Split processes that constitute a solver is left unspecified to a large extent. Instead, these processes are placeholders for software components that complement the functionality offered by the DICE framework. This component-based design is reflected by the configuration language. As an example, consider the statement

```
VARIABLE s IS DiscreteDomain "1..9";
```

$\langle Solver \rangle \rightarrow \langle Statement \rangle^+$

$\langle Statement \rangle \rightarrow$ `ENUM` $\langle Identifier \rangle$ `{` $\langle IdentifierList \rangle$ `} ;`
$\rightarrow$ `VARIABLE` $\langle Identifier \rangle$ `IS` $\langle Identifier \rangle$ $\langle Specifier \rangle$ `;`
$\rightarrow$ `VARIABLE` $\langle Identifier \rangle$ `IS` $\langle Identifier \rangle$ `ENUM` $\langle Identifier \rangle$ `;`
$\rightarrow$ `VARIABLE` $\langle Identifier \rangle$ `IS` $\langle Identifier \rangle$ `ENUM` $\langle Identifier \rangle$ $\langle Specifier \rangle$ `;`
$\rightarrow$ `DRF` $\langle Identifier \rangle$ $\langle Specifier \rangle$ `;`
$\rightarrow$ `WAIT;`
$\rightarrow$ `PRINT` $\langle Identifier \rangle$ `;`
$\rightarrow$ `SEARCH` $\langle Identifier \rangle$ $\langle Specifier \rangle$ $\langle Identifier \rangle$ $\langle Specifier \rangle$ `;`
$\rightarrow$ `SOLUTIONS;`

$\langle Specifier \rangle \rightarrow \langle QuotedString \rangle^+$
$\rightarrow$ `ENUM` $\langle QuotedString \rangle^+$

Fig. 2. Solver configuration language syntax

This creates a new Variable process. The identifier `s` is stored as a reference to this process. In the newly created process, a component in the "variable domain type" category is installed. The second identifier `DiscreteDomain` designates a particular component in that category. The specifier `"1..9"` has no meaning to the framework. It is assumed that the component identified by `DiscreteDomain` knows how to deal with it.

The `DRF` statement creates a DRF process and has similar semantics. Although the specifier strings have no meaning, any component in the domain reduction function category should be able to report the names of the variables that it applies to, when requested by the framework. The mechanism for this is described in the next section. This information is used to make the connections to the Variable processes. Propagation starts as soon as the first domain reduction function is installed. The `WAIT` statement activates the termination detection algorithm to postpone execution of the next statement until propagation has finished. The `PRINT` statement outputs the contents of a variable.

The first identifier-specifier pair of the `SEARCH` statement specifies the component installed in the Split process. The second pair concerns the Search process. This statement starts the search for solutions by creating and activating these processes. The solutions can be retrieved by means of the `SOLUTIONS` statement.

The `ENUM` keyword can be used as statement, to provide symbolic names for a range of integers starting at 0. The same symbolic name may not be used in more than one such statement. Also any specifier may be prefixed with `ENUM`. This effects a textual replacement of symbolic names by the corresponding integers. A third use of the `ENUM` keyword is in a `VARIABLE` statement. Here it is used to specify a set of symbolic names that should be used when printing integer values. In this case, textual replacement of symbolic names in the specifier is always carried out, and the specifier may be omitted to implicitly specify all values for which a symbolic name was provided.

4 Component Model

The interface between the DICE framework and the four categories of components is defined by four C++ abstract classes. As an example, consider the abstract class for components in the domain reduction function category:

```
class DRF_Component
{
public:
   static DRF_Component *UnitToDRF_Component( AP_Unit u );
   static DRF_Component *DescriptionToDRF_Component(
                            const char *type, const char *spec);
   virtual ~DRF_Component( );
   virtual AP_Unit ToUnit() const =0;
   virtual const std::vector<char*> &VariableNames( ) const =0;
   virtual int Compute( std::vector<Domain*> &arguments ) const =0;
   virtual int Termination( NodeType type,
                         std::vector<Domain*> &arguments );
};
```

In order to add a new component in any of the four categories, a subclass of the corresponding abstract class must be provided. In the case of the `DRF_ Component` example, the following provisions need to be made:

1. Through the `ToUnit` member function, objects in this class should be able to wrap themselves in a Manifold *unit*. By means of their unit representation, objects are transported through the process network.
2. Two static member functions `UnitToDRF_Component` and `DescriptionTo-DRF_Component` must be modified such that objects of the new class can be initialized from the unit representation generated by `ToUnit`, and from a textual representation, during parsing.
3. The actual functionality of the domain reduction function is embodied in the member functions `Compute` and `Termination`. `Compute` performs domain reduction as a part of the constraint propagation algorithm. `Termination` is called by the DICE framework upon termination of a single run of the propagation algorithm. Both member functions expect a vector of variable domain representations as an argument. The other argument to `Termination` indicates a solution, failure, or internal node. The `Domain` class is another abstract class, corresponding to the variable domain type component category.
4. In order for a DRF process to be able to relate the buffers associated with its input ports to the arguments of its domain reduction function, all `DRF_Component` objects should be able to supply a vector of variable names. The elements of this vector correspond to the elements of the argument vector of the `Compute` and `Termination` member functions.

Most of this work can be automated. In particular, a utility was written that generates the code for the `Compute` member function from rule-based specifications for operators on small finite domains [1].

Each of the Variable, DRF, Search, and Split processes of a solver executes a command loop. All coordination services are implemented by these processes exchanging commands. These commands are *tuples* of the Manifold language. As an example, consider the reduction request of Sect. 2.3. This command is implemented by the tuple

$$<\mathtt{CMD\_DRF\_REDUCE}, w, d>$$

where w and d are the unit representations of the world of the reduction request, and the new domain of the CSP variable in that world, respectively. The handler for `CMD_DRF_REDUCE` eventually invokes the `Compute` member function of the `DRF_Component` object installed in the receiving process.

5 Solver Building Blocks

In this section we describe a set of components that have been implemented in DICE to provide basic constraint solving capabilities.

Variable Domain Type

Components `IntegerInterval` and `DiscreteDomain` represent integer values by intervals and arbitrary sets of possible values, respectively. Recently, support for real numbers was also added to DICE by introducing a component `DoubleInterval` based on the JAIL library [6].

Domain Reduction Function

`IntegerArithmetic` and `DoubleArithmetic` components provide support for simple arithmetic on integers and reals. In both cases, the component instances are initialized from a string, representing a relation between two polynomials. `IntegerArithmetic` supports linear polynomials only. `DoubleArithmetic` is based on the interval arithmetic operators of JAIL.

`IntegerPairs` is initialized by two variable names, and a set of integer pairs that constrains the values that these variables may assume. For example:

```
ENUM Word5 {LASER,HOSES,SAILS,SHEET,STEER};
VARIABLE position1 IS DiscreteDomain ENUM Word5;
VARIABLE position2 IS DiscreteDomain ENUM Word5;
DRF IntegerPairs ENUM "position1,position2 IN"
   "(LASER,SAILS),(LASER,SHEET),(LASER,STEER),"
   "(HOSES,SAILS),(HOSES,SHEET),(HOSES,STEER)";
```

The `MultiDRF` domain reduction function takes as an argument a sequence of DRF statements. For example:

```
DRF MultiDRF
   'DRF IntegerArithmetic "q2 != q7";'
   'DRF IntegerArithmetic "q2 - q7 != 5";'
   'DRF IntegerArithmetic "q2 - q7 != -5";'
;
```

The purpose of this component is to group several domain reduction functions into a single process of the distributed solver[2]. The `Compute` member function of a `MultiDRF` computes a fixpoint of its constituent domain reduction functions, by repeatedly applying these functions in sequence until a full sequence passes in which no changes to a variable domain are made.

As an alternative to having special components that implement optimizing search strategies, such as branch-and-bound, a `Minimize` component is available, that can be used to transform any search strategy into an optimization strategy. Figure 3 illustrates the use of this component. An additional variable is constrained such that its domain contains the outcome of an objective function. `Minimize::Termination` records the smallest value (singleton set) for this variable, encountered for a solution so far. `Minimize::Compute` imposes a dynamic constraint, stating that only improvements of this bound will be considered as new solutions.

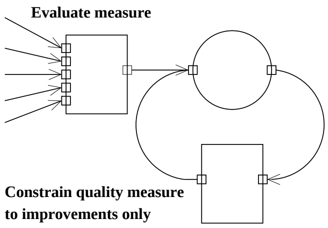

Fig. 3. The approach to optimization encouraged by DICE

Splitting Method

Currently, only one splitting method implementation exists, that splits a variable domain according to the default splitting method for its domain type. For `DiscreteDomain` the default is to generate a subdomain of size one for each element of the original domain. For `IntegerInterval` and `DoubleInterval` the default is bisection. The variable selection method is "dynamic fail-first," selecting a variable with the smallest number of alternatives at each internal node.

Search Strategy

Three search strategies have been implemented:

1. A search strategy that reconstructs the search tree on the basis of the textual representation of the worlds that it learns about. When propagation finishes in an internal node, it waits for new worlds to become available. In a leaf

[2] It is also possible to group several variables into a single process.

node, or when new worlds are reported by a variable, the next world is selected according to a chronological backtracking strategy.
2. A search strategy that puts all new worlds in a queue. When new work is required by the propagation network, the world at the front of the queue is selected. This behaves in a breadth-first fashion.
3. A similar strategy, but employing a stack, instead of a queue. This strategy behaves in a depth-first fashion.

The latter two strategies have a target number of worlds where propagation is carried out at the same time. They keep injecting work into the network until this target is met.

Work in Progress

The set of components discussed in this section can be used to demonstrate the effectiveness of DICE on many standard examples. However, much work still needs to be done before we can claim to have a useful constraint solver. For example, our integer arithmetic presently cannot multiply variables, our constraints over the reals are currently limited to quadratic equations, our default split method does not yet properly take into account domain sizes of reals, and it is not yet possible to relate integer and real variables.

Also, in order to use DICE as a platform for experimenting with different search strategies, alternatives must be provided in the splitting method and the search strategy component categories:

- chronological, static and dynamic fail-first, and maximum cardinality (most constrained first) variable selection methods,
- enumeration, bisection and middle-first value selection methods,
- limited discrepancy search.

Likely, the abstract classes for the component categories will still change. For example, it seems desirable to make the distinction between variable selection and value selection in splitting explicit. Also, there are many similarities between DRF and Split processes, and it may make sense to integrate the two component categories to indicate that splitting is a special case of domain reduction.

Currently, complex domain reduction functions can be configured from atomic reduction steps by means of the `MultiDRF` component. We plan to provide similar composition operators in the other component categories. Imagine a composite splitting method that dynamically changes from one basic splitting method to another, as search progresses. Other components that we plan to implement are:

- a box-consistency enforcer for constraints over the reals;
- domain reduction functions that take a set of constraints as a specifier and enforce these constraints inside a single reduction step by means of well-known propagation algorithms, such as AC-3 and AC-4;
- a domain reduction function that constrains an integer variable to count the number of violations in a set of soft constraints. Together with the `Minimize` component, this can be used to implement Max-CSP.

6 Discussion and Directions for Future Work

Many libraries, frameworks and toolkits for constraint solving exist. DICE is an implementation of the ideas presented in [2, 8], and is distinguished from other platforms by the combination of an open-ended and inherently parallel design.

An example of a (non-parallel) system that is open-ended in a similar way is Figaro [7]. Here also new components for solver configuration can be added by inheriting from abstract classes corresponding to a number of (different) solver design dimensions. Often such systems are libraries. Inside the same application source we may both code new solver components, and employ the solvers constructed from these and other, readily available components. While at some point it will be desirable to provide an API to DICE, adding new components and using DICE for constraint solving are two distinct matters by design. This emphasizes that DICE is intended as a workbench environment, where constraint solvers are configured from basic building blocks. As a consequence, the application programmer is shielded from the Manifold system to a large extent. Only the component programmer is aware, for example, that components should be able to wrap themselves into Manifold units.

It could be argued that this scheme fails for components generated from a rule-based definition of reduction operators. These will be application specific, and should be part of the application source instead of being installed in DICE. A solution would be to provide a generic component, to which the rules are submitted as a specifier.

For related work on the level of the algorithm of Sect. 2.3, the reader is referred to previous work on chaotic iteration for distributed constraint propagation cited in [8]. It should be noted that the motivation for distributed processing is the expectation that efficiency can be gained from solver cooperation. While many common issues exist, this starting point is essentially different from the assumptions in the area of distributed constraint satisfaction *problems*, of which an overview is given in [10]. The algorithms in that reference tackle the problem of having a CSP solved by a number of *agents*, each of which knows part of the problem, while it is impossible or undesirable for any one agent to gather all constraints. In this scenario, using facilities of the IWIM model for external coordination would mean a loss of generality.

The set of components described in the previous section can be used to demonstrate the effectiveness of the framework, but does not yield efficient solvers, and at this point we have no reasons for running benchmarks. This is because in the current implementation, DICE focuses on parallelism on the level of individual reduction steps only. Reduction of turn-around time by exploiting parallelism should be on a sufficiently large granularity of individual tasks, and in general, a reduction step is too small to justify the overhead of coordination. Yet the possibility to have a reduction step running in its own Manifold process makes it easy to interface with external solvers, possibly running on dedicated machines. We plan to investigate solver cooperation schemes, as discussed in [9], that can benefit from these facilities.

A more straightforward way to achieve larger task granularity is to add support for or-parallelism. Search strategies that perform propagation in multiple nodes of the search tree at the same time are already available in DICE, but the propagation engine employs one process for each reduction step, and multiplexes the work in different worlds, to a large extent. One way to increase the capacity of the propagation network is to add an extra level of coordination, and replace the DRF processes by coordinators managing a number of DRF processes that handle the actual reduction requests in parallel.

In many cases, however, the size of the search tree will be too large to justify even having communication involved with every node of the search tree. Therefore support should be added for components that can autonomously explore a part of the search space. This can be seen as a form of splitting method. A full solver can be incorporated as a splitting method that generates a new world for each solution, and possibly several worlds for the parts of the search space that still need to be explored. This way, splitting becomes a compute-intensive task, and exploiting or-parallelism becomes a matter of coordinating multiple Split processes. The distributed constraint propagation algorithm can then be used for optimization among the results of these parallel solvers.

References

1. K.R. Apt, E. Monfroy. Automatic Generation of Constraint Propagation Algorithms for Small Finite Domains. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, LNCS 1713, pp. 58–72, Springer Verlag, 1999.
2. F. Arbab, E. Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In Porto, Roman (eds.) *Coordination Languages and Models*, LNCS 1906, pp. 115-132, Springer Verlag, 2000.
3. F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In Ciancarini, Hankin (eds.) *Coordination Languages and Models*, LNCS 1061, pp. 34–56, Springer Verlag, April 1996.
4. F. Arbab. *Manifold version 2: Language reference manual*, Technical report, Centrum voor Wiskunde en Informatica. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z.
5. E.W. Dijkstra. *Shmuel Safra's Version of Termination Detection* (Note EWD998).
6. F. Goualard. *JAIL - Just Another Interval Library*, Manual rev. 0.0.9, Institut de Recherche en Informatique de Nantes, January 2002.
7. M. Henz, T. Müller, K.B. Ng. Figaro: Yet Another Constraint Programming Library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, held in conjunction with ICLP'99.
8. E. Monfroy. A Coordination-based Chaotic Iteration Algorithm for Constraint Propagation. In Carroll, Damiani, Haddad, Oppenheim (eds.) *Proceedings of the 2000 ACM Symposium on Applied Computing*, pp. 262–269, ACM Press.
9. E. Monfroy, F. Arbab. Constraints Solving as the Coordination of Inference Engines, in Omicini, Zambonelli, Klusch, Tolksdorf (eds.) *Coordination of Internet Agents: Models, Technologies and Applications*, Springer Verlag, 2001.
10. M. Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Springer, 2001.