Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):
Chapter 9

Distributed Constraint Solving

The subject of this chapter is DICE (Distributed Constraint Environment), a framework for distributed constraint solving. In addition to the design of the framework, and its implementation in the Manifold coordination language, we will discuss a number of extensions that were proposed to improve the efficiency of distributed constraint solving in DICE. OpenSolver was originally developed as a part of DICE, to realize these optimizations, and the material presented here clarifies some of the design decisions. It also demonstrates a third potential application of OpenSolver as a software component.

9.1 Introduction

Constraint propagation algorithms can essentially be characterized as a set of functions, plus a scheduler that coordinates their application. This supports the observation of Gelernter and Carriero that all useful programs consist of a combination of computation and coordination [GC92]. This observation was made in the context of coordination languages. Although they originated in the area of parallel and distributed computing, coordination languages now also manifest themselves as a technology for realizing component-based software engineering.

As we have seen in the previous chapters, constraint solving comprises a collection of largely independent techniques. To solve a CSP efficiently, a constraint solver typically combines several procedures, heuristics, and even stand-alone solvers. This indicates that in addition to providing evidence for the “programming = computation + coordination” proposition, constraint solving might also benefit from the component-based approach that is facilitated by contemporary coordination languages.

The above observations suggest a coordination-based implementation of constraint solving, which was explored in two articles: [Mon00a], on a coordination-based constraint propagation algorithm, and [AM00], which complements this algorithm with facilities for search in a distributed setting, i.e., in absence of a
centralized representation of CSPs. An additional benefit of the approach proposed in these articles is that because of the concurrent nature of the coordination language, the solver can be applied in situations that require distributed solving, or where it can be expected that parallel execution of domain reduction functions will reduce the turn-around time of solving.

An implementation of the distributed constraint propagation algorithm in the Manifold coordination language provided the proof of concept, and DICE [Zoe03b] combined both algorithms into a general purpose coordination-based constraint solver. Because of its distributed nature (every variable and reduction operator in DICE had its own thread, possibly running in its own process), constraint solving in DICE involved massive communication among concurrent threads, which made it quite inefficient compared to existing, sequential constraint solvers. For this reason, in [Zoe03a] we proposed an alternative implementation that allows an arbitrary distribution of variables and reduction operators over a set of cooperating solvers. The resulting system allows all configurations ranging from a fully distributed solver to a single sequential constraint solver. OpenSolver was intended to implement the cooperating solvers of this alternative implementation.

The remainder of this Chapter is organized as follows. Section 9.2 covers the original DICE system. It briefly introduces the Manifold coordination language, and recalls the coordination-based constraint solving algorithms of [Mon00a] and [AM00]. In Section 9.3 we describe the alternative implementation proposed in [Zoe03a], and in Section 9.4 we relate this proposed alternative to the current OpenSolver implementation. In Section 9.5 we evaluate the benefits of our approach, and discuss related work.

9.2 DICE

DICE (Distributed Constraint Environment) is a framework for distributed constraint solving, implemented using the Manifold coordination language. A running system consists of a number of processes, that cooperate according to coordination protocols for constraint propagation, distributed termination detection, and search. These are described below in Sections 9.2.2–9.2.4. First we introduce Manifold and its coordination model.

9.2.1 Coordination Model and Language

Coordination languages offer language support for composing and controlling software architectures made of concurrently executing entities. In the Idealized Worker Idealized Manager (IWIM) model of coordination [Arb96], these entities 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 event 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 [Arb96, Arb] 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.

### 9.2.2 A Distributed Constraint Propagation Algorithm

In contrast to inherently sequential constraint propagation algorithms like Algorithm 2.1 on page 22, DICE implements the coordination-based chaotic iteration algorithm of [Mon00a]. In this algorithm, each CSP variable is represented by a process that maintains the domain of that 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. Channel connections are made between the ports of Variable and DRF processes according to the structure of the CSP. The DRF processes have a buffer associated with each input port, which stores the domain last seen on that port. These buffers are initialized by having a Variable process send its domain each time a connection to a DRF process is made.

Figure 9.1(a) shows an example process network of this algorithm. Variable processes send reduction requests to DRF processes. 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.

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 current domain, 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.

In [Mon00a] it is argued that this distributed algorithm implements a restriction of the Generic Iteration Algorithm for Compound Domains of [Apt99]. This allows us to benefit from several properties that have been proven for that algorithm. One of these properties is that the algorithm is guaranteed to terminate if the domains are finite and the DRFs are inflationary. The latter is effectively ensured by having Variable processes compute intersections.

The elegance of the coordination-based algorithm is that neither the Variable processes nor the DRF processes have to know anything about the CSP that they are solving. The input and output schemes of DRFs, which are used in Algorithm 2.1 to maintain the set \( G \) of DRFs that still need to be applied, are

\(^{1}\)In the implementation, 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.
9.2.3 Distributed Termination Detection

Although this is not strictly necessary, usually we do not want to consider expanding the search tree by branching on the domain of a variable before constraint propagation has finished. Therefore, we need to know when the propagation algorithm terminates. With a sequential algorithm, like Algorithm 2.1, 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 [Dij87] 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 Figure 9.1(a) show the extra channels for termination detection. All processes maintain a local counter of the number of update commands and reduction requests in the network. The ring network is used for circulating a token. This token is forwarded when the process that holds it becomes idle. When it returns to the process that created it, the token has accumulated the local counters of all process. Termination can be concluded only if this sum equals 0. Together with a black/white coloring of the processes and the token the algorithm ensures correctness in case of asynchronous channels. This corresponds to the Manifold communication model.

9.2.4 Search

DICE employs a scheme similar to that of [AM00], where the network of processes of the constraint propagation algorithm is performing work in several nodes of the search tree simultaneously. As a result, multiple tokens of the termination detection algorithm may be circulating on the ring network, one for every instance of the constraint propagation algorithm, and 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:
Variable::
\[
\begin{align*}
  v: & \text{World} \overset{m}{\rightarrow} \text{Domain} \\
  \text{color}: & \text{World} \overset{m}{\rightarrow} \{\text{black, white}\} \\
  n_{\text{msg}}: & \text{World} \overset{m}{\rightarrow} \mathbb{Z}
\end{align*}
\]

DRF::
\[
\begin{align*}
  I: & \text{ARRAY} [1..n] \text{ OF World} \overset{m}{\rightarrow} \text{Domain} \\
  \text{color}: & \text{World} \overset{m}{\rightarrow} \{\text{black, white}\} \\
  n_{\text{msg}}: & \text{World} \overset{m}{\rightarrow} \mathbb{Z}
\end{align*}
\]

where \( n \) is the number of input ports of the DRF process, and \( \text{World} \) is a datatype whose elements serve as identifiers for nodes of the search tree. \( A \overset{m}{\rightarrow} B \) denotes a map data structure, i.e., a set of tuples \( \langle a, b \rangle \in A \times B \) in which every \( a \in A \) occurs at most once. Maps \( v \) and \( I[1], \ldots, I[n] \) hold the data for the propagation algorithm, and \( \text{color} \) and \( n_{\text{msg}} \) represent the state of the termination detection algorithm.

A partial order is defined on the elements of \( \text{World} \), by which an ancestor node is compatible to its descendants, and 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 \textbf{Split} and \textbf{Search}, which implement the branching strategy (involving variable selection and value selection) and traversal strategy, respectively. These processes have connections to all Variable processes (Figure 9.1(b)), and coordinate the network of the propagation algorithm to perform search.

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 branch on (if any), and construct a set of new \( \text{World-Domain} \) pairs for that variable. The worlds of this set correspond to the subproblems created by the branching.

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 it that new worlds have become available.
9.3. Cooperating Solvers

The design of the previous section supports the cooperation of solvers on the level of reduction steps inside the branch-and-propagate search. Because of the very small grain size of the computational tasks that are typically performed in a reduction step, this has limited applicability. Therefore, in [Zoe03a], we proposed to adopt a more general scheme, which is comparable to that of [MR99], from a constraint propagation point of view. The basic process instance in this scheme is a solver. A solver process can:

- maintain any number of variables,
- apply DRFs to variables that it maintains,
- branch on the domains of variables that it maintains in order to generate new nodes in the search tree, and
- start constraint propagation in nodes of the search tree that it knows about.

In Sections 9.3.1 and 9.3.2 we discuss some details and implications of this scheme, related to constraint propagation and search, respectively. In Section 9.3.3 we introduce DRF worker processes to support parallel search. Figure 9.2 shows an example network of solvers.

9.3.1 Grouping Variables and Reduction Operators

Solver processes can have a pair (input and output) of ports for each of the variables that they maintain. Channel connections can be made to these ports in order to connect variables in solver processes that correspond to the same CSP variable. Solvers modify the domains of CSP variables by computing a fixed point of their local DRFs. When a solver process modifies the domain of a variable that it maintains, and for this variable there exists an output port
that has one or more channel connections, the new domain for that variable is sent through this output port as a *propagate command*. Propagate commands are handled like the update commands and reduction requests of Section 9.2.2. Incoming propagate commands that reduce the domain of a CSP variable trigger the fixed point computation.

Compared to the design of Section 9.2.2, we can now combine several Variable processes into a single process. Also domain reduction functions that involve the corresponding CSP variables can be applied by this same process directly, without communication. Solvers can have local variables, for which no ports and channel connections exist. This way, communication takes place only for CSP variables that are shared by solvers. Based on the results reported in [MR99], we can expect that for sufficiently large problems, a partitioning of CSP variables and DRFs can be found for which efficiency can be gained from distributed execution.

The DRF processes of Section 9.2.2 are special cases of solvers that apply a single DRF. The Variable processes can be implemented as solvers that maintain a single CSP variable, and do not apply any DRFs. In the original design, Variable processes served to coordinate the activities of the DRF processes by forwarding update commands as reduction requests. In DICE, solvers can send each other updates of variable domains directly. Network topology is no longer centered around a set of processes whose main task is to forward updated variable domains. Failing to connect two solvers on a common CSP variable, however, may influence the level of consistency that is enforced by constraint propagation. Therefore we should provide a default topology that ensures the maximum level of consistency that can be achieved by the domain reduction functions.

### 9.3.2 Search by Cooperating Solvers

More than one branching strategy may be active in a network of solver processes. This has two potential applications:

- **Complementary strategies** that cooperate to implement a global strategy. Every solution occurs exactly once in the global search tree. The obvious example here is that different solvers branch on different variables.

- **Competing strategies**, where there are different ways of arriving at the same solution. This can be useful when searching for a single solution. This is also sometimes called *diversification*.

For complementary strategies, facilities must be provided that allow cooperating solvers to adhere to a common variable selection strategy. For example, if indeed we use a dedicated process for each variable, as in Section 9.2.2, it makes sense to let these processes control the branching of their variables. In that case, to implement fail-first, these processes must be able to determine among themselves which process holds the variable with the smallest domain. In the presence
of competing strategies, in order to control the size of the search tree, we will probably want to prevent one strategy from splitting a subproblem generated by another strategy. For this purpose, nodes of the search tree generated according to different strategies must be distinguishable, and form separate subtrees of the global search tree.

New nodes that are generated inside a solver by application of a branching strategy can be handled in two ways:

- they can be stored locally, in a set of nodes that await constraint propagation, or
- they can be sent to another solver via dedicated ports and channels.

The purpose of the latter option is to allow one solver to coordinate the traversal of the search tree, in the case that more than one solver is able to generate new nodes. This solver then plays the role of the Search process of Section 9.2.4. Reports of new nodes created by another solver are treated by the receiving solver in the same way as new nodes created internally. If a node is labeled with the solver process and branching strategy that created it, it is always known what other solver needs to be instructed to start propagation in a particular node of the search tree.

Solver processes consult their traversal strategy plug-ins, if available, on two occasions: (1) when constraint propagation terminates in some node of the search tree, and (2) when new nodes become available in the solver (created internally, or reported by another solver). On these occasions propagation may be started in any node of the search tree that the solver knows about. A special instance of the termination detection algorithm is needed to detect termination of the global search, by counting the nodes of the search tree that await constraint propagation.

Compared to the scheme of Section 9.2.4, where the traversal is coordinated from outside the propagation network by the Search process, we now have the option to let this be handled by the solvers that are performing constraint propagation. From one point of view this can be regarded as mixing concerns that were separated in the IWIM design. From another point of view, it can be explained as a looser form of coordination: in principle propagation is performed as soon as a new node of the search tree becomes available. But to regulate the traffic in the network, the processes may choose to hold the messages in several nodes of the search tree, and release the messages in others, as bandwidth becomes available. Internally this is implemented by keeping a set of nodes that await propagation, and selecting nodes from this set according to a traversal strategy.

### 9.3.3 Parallel Search by Delegation

As a second extension to the design of Sections 9.2.2–9.2.4, solver processes are allowed to delegate the actual application of domain reduction functions and
branching strategies (internally these have a common interface) to DRF worker processes in a master-slave fashion. Using this option, the solvers become IWIM managers themselves. The main reason for introducing this extra level of coordination is to provide support for parallel search. Constraint propagation may already be running for several nodes of the search tree simultaneously, but with only one process instance available for each solver, the network will be multiplexing the work in these different nodes, to a large extent. On the one hand, a pool of DRF workers increases the capacity of the network to actually handle the propagate commands in different nodes in parallel.

On the other hand, at the task granularity of a single reduction step, the communication overhead will generally outweigh the potential gain from exploiting parallelism, and there is little justification for doing this. This facility is useful only for computation-intensive reduction steps. The primary example would be a solver that autonomously explores a subtree, and splits the root node of this subtree into a set of nodes that contains a leaf node for every solution, plus several internal nodes for the part of the subtree that it has not yet explored.

When using DRF workers, for the purpose of the termination detection algorithm a solver is considered to be idle in a particular node of the search tree when (1) no commands (concerning any node) are immediately available on any of its input ports, (2) there is no need to compute the fixed point of the DRFs for that node, and (3) the solver is not expecting any results from DRF workers concerning that node of the search tree. Many options still exist for implementing this coordination pattern. In particular, we propose to use a pool of DRF workers per solver, and not to have DRF workers cache variable domains between two calls. This involves more communication than necessary, but this should not be a problem for the coordination of autonomous solvers, as suggested above.

### 9.4 Implementation

A full implementation of the DICE system, as described in Section 9.2, exists. It is implemented using Manifold, with atomic workers written in C++. The DICE implementation has a plug-in system comparable to that of OpenSolver, with only four categories: variable domain types, domain reduction functions, branching strategies, and traversal strategies. In the context of DICE, the plug-ins are called *components*. A more detailed description can be found in [Zoe03b]. As an optimization, several DICE variables can be combined in a single process, and also an adapter-like domain reduction function exists that computes a common fixed point of a set of other DRFs. This gives some control over the amount of communication in the constraint propagation phase, but cannot prevent that data is exchanged for every node of the search tree. Given the size of the search tree even for problems that can be solved within a few seconds (see for example, Table 7.1 on page 162), this is still an obstacle for competitive performance.
9.4. Implementation

OpenSolver was intended to play the role of the cooperating solvers of Section 9.3. In this role it would be complemented with a coordination layer plug-in that offers the set of ports shown in Figure 9.3. These ports can then be connected by Manifold channels to form networks like that of Figure 9.2. This intended use has greatly influenced the design of OpenSolver, presented in Chapter 3. It explains several features that do not follow directly from the model of constraint solving of Section 2.3, notably

- the world database,
- maintaining a set of nodes that are subject to constraint propagation, and
- the coordination layer, and giving it responsibilities like the final confirmation that constraint propagation has terminated.

Although OpenSolver was designed to implement the cooperating solvers of the previous section, this system was never fully implemented, and some aspects of the design are still missing. The facilities described in Section 3.3.2 have been implemented only to the extent that nested search and parallel search are supported. Notably the commands pending sends and update, and the bookkeeping of variable changes for exporting modified domains have not been implemented. Furthermore, neither the set of commands of Section 3.3.2, nor the current implementation supports delegating the application of domain reduction functions to DRF worker processes, as proposed in Section 9.3.3.

Having said this, the design of OpenSolver does support an easy implementation of these features, and after implementing the distributed solver described in Section 9.2, we are confident that the design is well suited for distributed constraint propagation. The results in Chapter 8 also essentially prove the concept of parallel search by delegation, so technically, the distributed solver of the previous section is feasible, but a full implementation was not needed for our experiments.
9.5 Discussion

9.5.1 Benefits

From our experience with the DICE system we can conclude that the design of Sections 9.2.2-9.2.4 leads to an effective general-purpose distributed constraint solver. Given the efficiency of OpenSolver, which was demonstrated on several benchmark problems in the previous chapters, we can expect that the optimizations proposed in Section 9.3 lead to an efficient implementation of the system, allowing local processing where distribution is not required. Nevertheless, we should be careful in the assessment of the benefits of our approach. We discuss these on the basis of the following aspects: the role of coordination, the need for distributed constraint solving, and the contribution of a system that combines parallel and distributed processing.

The Role of Coordination

As we discussed in Section 9.1, constraint propagation through iteration of domain reduction functions can be seen as a form of exogenous coordination. This is made explicit in the IWIM design of the constraint propagation algorithm of [Mon00a], which we used in Section 9.2.2, where the processes that apply the DRFs are coordinated by a layer of processes corresponding to the variables of a CSP. Both kinds of processes are atomic workers in the sense that they are largely unaware of the environment that they operate in. The computation that they perform is determined by the network of channels that connects these processes. In this sense, the algorithm can be characterized as a coordination-based algorithm, but in our opinion, it is primarily a distributed algorithm. Its conformance to the IWIM coordination model is an advantage only in comparison with other distributed algorithms. For example in a distributed algorithm that relies on message passing, the processes would need to know the identification of the processes that they communicate with. Compared to such algorithms, the coordination-based approach is more elegant and flexible, and can more easily be verified to be correct.

Since we use a distributed algorithm for constraint propagation, we have to deal with distributed termination detection, and coordination models and languages could also be of importance here. In the current model and implementation, the automatic replication of messages through ports with multiple outgoing channels implies that for counting the number of messages in the network, which is inherent to many termination detection algorithms, the variables need to know how many DRFs they are connected to. This runs counter to the inherent simplicity of the constraint propagation algorithm proper. For example, we now have to temporarily halt a Variable processes before making or removing a channel connection to its output port, in order to correctly update the counter of its outgoing
9.5. Discussion

channels before it sends any new messages. Because termination detection is such a common problem, in our opinion, primitives for it would be valuable additions to a coordination language like Manifold.

Even without support for termination detection, using a proper coordination language like Manifold is convenient because it has rich facilities for process management, and for channel-based communication. However, it does not simplify composing constraint solvers from software components, as we anticipated at the beginning of this chapter. This has two reasons.

- As we discussed in Section 3.3.3, from a coordination point of view, a software component typically has its own thread of control and interacts with its environment through a set of ports. This model does not fit the majority of constraint solver building blocks that we categorized in Chapter 3, and wrapping them up as autonomous processes is artificial.

Conversely, in those cases where we want to use an autonomous solver that fits the model of an IWIM process, a component-based framework like OpenSolver can communicate with it through a proxy. Wrapping up autonomous solvers as objects through proxies is equally artificial, but at least it entails that the appropriate method of software composition is used for the majority of units of composition. In other words, do not distribute everything because in a few cases, this makes software composition easier.

- While some specific instances of these building blocks are of a complexity that calls for code re-use, most of them entail very simple data structures and algorithms. The effort of implementing a configurable constraint solver is in the definition of their interfaces, rather than in writing the code for these data structures and algorithms. Conformance to the IWIM model, or the use of a coordination language does not simplify this task.

In summary, if for some reason it is necessary to perform distributed constraint propagation, the algorithm of [Mon00a] provides an effective solution. Because of its conformance to the IWIM coordination model it has specific advantages over other solutions to distributed constraint propagation, and it is convenient to use a coordination language like Manifold to implement it.

The Need for Distributed Constraint Propagation

Because communication is involved with domain updates, distributed constraint propagation is communication-intensive. In general, communication will outweigh computation, and we can expect to be able to apply it efficiently only in the following cases:

- In case of very computation-intensive reduction operators we may achieve a reduction of turn-around time if the cooperating solvers run on parallel processors. The operator of Section 7.3.1 easily involves seconds of computation
time per application, and would be a good example. For our application we need precisely one instance, but there probably exist situations where it would make sense to use multiple operators of comparable functionality.

- If the problem itself is distributed, while it is impossible or undesirable to gather all information in a single location, distributed constraint propagation is unavoidable if we want to perform a branch-and-propagate tree search.

In the first case, if constraint propagation is performed as a part of branch-and-propagate tree search, it will likely be more efficient yet to parallelize the search instead of the propagation. The latter case is known as the distributed constraint satisfaction problem (DisCSP). Our approach is definitely suitable for DisCSP solving.

Combining Parallel and Distributed Constraint Solving

The extensions proposed in Section 9.3.1 are a valuable addition to the design of Sections 9.2.2–9.2.4. It allows us to group variables and DRFs such that communication is limited to what is absolutely necessary for a given DisCSP situation. In addition, in Section 9.3.3, we proposed to increase the capacity of the constraint propagation network by delegating the actual application of domain reduction functions to DRF worker processes, with the particular goal to support parallel search. In retrospect, it does not seem likely that parallel search and distributed propagation need ever be combined, and there is no need for a system that supports both. The coordination-based approach of Chapter 8 is much simpler and more flexible, and the design of OpenSolver is complicated enough without the facilities for delegation.

Another justification for the delegation mechanism would be that it prevents that time-consuming constraint propagation in one node of the search tree prevents progress of the search in other subtrees, that are being explored simultaneously. In our opinion, the option to have an OpenSolver scheduler plug-in return control before a fixed point has been computed, as discussed in Section 3.2.3, is a better solution.

Finally, since we want to prevent that a branching strategy is applied to a node that is generated by another branching strategy, the competing branching strategies of Section 9.3.2 can also be implemented by having multiple solvers running concurrently. For optimization purposes, we can use the time-out mechanism of Chapter 8, and share bounds implied by new suboptimal solutions between multiple instances of loop networks as shown in Figure 8.6. If the number of available processors is smaller than the number of competing strategies, these loops should then contain only one solver. For competing strategies, the advantage of the interrupt mechanism discussed in Section 8.6 over the time-out mechanism is even
greater, because it allows that new bounds are taken into account immediately, without having to wait for the elapse of a time-out.

## 9.5.2 Related Work

Above we argued that we should not distribute a constraint propagation algorithm based on generic iteration, just to provide for the case that one of the DRFs that we want to apply happens to be an autonomous process. BALI [Mon00b] is a system that supports exactly this mode of solver cooperation. Fixed point computation is just one of the cooperation patterns of BALI, and likely, BALI itself could be implemented as an instance of a (distributed) generic iteration algorithm. From this perspective, it is all a matter of selecting the right method of software composition for the units that we want to combine. A Manifold implementation of BALI is investigated in [AM98].

In our opinion, the coordination-based approach is beneficial only in comparison to other distributed constraint solvers. In particular, the distributed constraint propagation algorithm of [MR99] is mentioned in [Mon00a].

In branch-and-propagate tree search, even in the case that different parts of the search tree are explored in parallel, the expansion of the search tree by branching and the selection of the nodes for further exploration are inherently synchronous and sequential operations. In contrast, the \textit{asynchronous backtracking} algorithm distributes the search itself. Different \textit{agents}, each maintaining their own variable, propose values for these variables to other agents, with whom they share a constraint. By exchanging such proposals and no-goods, the agents will eventually find a solution if it exists. An overview of asynchronous backtracking and related algorithms is given in [Yok01]. These algorithms still rely on distributed termination detection, but now it has to be established only once, to detect that consensus has been reached. The Disolver system [Ham05] is reported to support this kind of distributed search. It would be interesting to compare the performance of both approaches, asynchronous backtracking and that of Sections 9.2.2–9.2.4 with respect to communication, CPU time, turn-around time, and memory usage. To our knowledge, such a comparison has not been made.

Diversification, i.e., competing search strategies, in the context of distributed search are discussed in [RH05]. A variant of the DisCSP problem where constraints can also be retracted is known as the \textit{dynamic} distributed constraint satisfaction problem (DynDCSP). An algorithm for maintaining arc consistency for this class of problems is given in [Rin05]. OpenSolver cannot deal with constraint retraction, and our approach to constraint solving in general is unsuited for solving of dynamic CSPs.

A different approach to exploiting parallelism in the constraint propagation phase is reported in [GH00]. This approach involves an alternative iteration algorithm that applies the following steps until a fixed point of all reduction
operators is reached.

1. Apply all pending reduction operators independently of each other to evaluate the amount of reduction they achieve on the current set of domains.

2. Compute a fixed point only of those operators that achieve the maximal reduction.

The first of these two steps can easily be parallelized because the operators are all applied independently of each other, on the same set of domains. The second step is computed sequentially, and constitutes a bottleneck for parallel performance. As a result, only modest speedup figures are obtained beyond running the alternative iteration algorithm on a single processor.

Finally, we would like to mention that **concurrent constraint programming** (CCP, see, e.g., [SR90]), is a model of concurrent computation, where processes or agents interact by communicating with a shared constraint store. The communication with the store consists of **Ask** and **Tell** operations. Via the Ask operation, an agent inquires whether a constraint is entailed by the store or not. Via the Tell operator constraints are added to the store. Although distributed computing introduces concurrency, the algorithms discussed in this chapter are unrelated to CCP. We expect, however, that they can be used to realize a distributed implementation of a CCP constraint store, if needed. The multi-paradigm programming language Oz (see, e.g., [VRBD+03]) incorporates the CCP model. Its main implementation, Mozart (see [http://www.mozart-oz.org/](http://www.mozart-oz.org/)), is an advanced platform for the development of distributed applications.

### 9.6 Summary

In this chapter we presented the DICE framework for distributed constraint solving, and discussed an optimization of it, which entails combining the functionality of several processes in order to limit communication overhead. Through a special coordination-layer plug-in, OpenSolver can be configured to implement this optimization. The resulting system is well suited for solving the DisCSP problem, and conformance to the IWIM model makes it a very flexible solution.

We also considered adapting OpenSolver/DICE for parallel search and competing search strategies. We argued that these are better dealt with according to the component-based approach of Chapter 8. In retrospect, we should not modify a software component to support functionality that can also be achieved by exogenous coordination of these components.