Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

A coordination-based framework for distributed constraint solving

P. Zoeteweij

REPORT SEN-R0230 DECEMBER 31, 2002

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

A Coordination-Based Framework for Distributed Constraint Solving

Peter Zoeteweij
email: P.Zoeteweij@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
DICE (DIstributed Constraint Environment) is a framework for the construction of distributed constraint solvers from software components in a number of predefined categories. The framework is implemented using the Manifold coordination language, and coordinates the components of a distributed solver by means of coordination protocols that implement a distributed constraint propagation algorithm, a distributed termination detection algorithm, and facilities for the distributed splitting of a constraint satisfaction problem. The aim of this report is to give an overview of DICE. In addition to the coordination protocols, the report describes the framework architecture and solver configuration language, and discusses our plans for further development.

1998 ACM Computing Classification System: D.1.3 [Programming Techniques] Concurrent Programming, D.1.m [Programming Techniques] Constraint solving, D.2.11 [Software Architectures] Domain-specific architectures, D.2.6 [Programming Environments] Programmer workbench.
Keywords and Phrases: Distributed constraint solving, cooperative constraint solving, coordination, IWIM, Manifold, component-based software.
Note: This work is supported by NWO under project number 612.069.003.
Note: This report combines references [15, 16, 17].

1. Introduction

The applicability of the Idealized Worker Idealized Manager (IWIM) model of coordination in the area of constraint solving has been studied in [3, 11]. This work has materialized in DICE (DIstributed Constraint Environment), a framework, implemented using the Manifold coordination language, where distributed constraint solvers are constructed from software components in a number of predefined categories:

- domain types for the variables of a constraint satisfaction problem (CSP),
- (incomplete) constraint solvers that can act as domain reduction operators inside a constraint propagation algorithm,
- splitting strategies, which (in combination with the pruning capabilities of the incomplete solver components) define a search tree, and
- search (traversal) strategies, which determine how to explore this search tree.

Each component instance can run in a separate process, and the framework coordinates the activities of the components by means of existing coordination protocols that implement a distributed constraint propagation algorithm, a distributed termination detection algorithm, and facilities that allow a splitting strategy component and a search strategy component to coordinate the network of processes to perform search. The constraint propagation algorithm applies the incomplete solvers until none of these can reduce the problem any further. In general, this will not lead to a solution

for 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 before constraint propagation is finished.

The aim of this report is to give an overview of DICE. In Section 2 we summarize the relevant aspects of constraint solving, coordination programming, the IWIM model, and Manifold. In Section 3 we recall the coordination protocols. In Section 4 we discuss two extensions that we plan to implement: the grouping of related components into cooperating solvers, and support for parallel search. Implementation is discussed in Section 5. In Section 6 we discuss several possible applications of the target framework.

2. Preliminaries

2.1 Constraint Solving

Constraint solving deals with finding solutions to constraint satisfaction problems. A CSP consists of a number of variables and their associated domains (sets of possible values), and a set of constraints. A constraint is defined on a subset of the variables, and restricts the combinations of values that these variables may assume. A solution to a CSP is an assignment of values to variables that satisfies all constraints. DICE supports two techniques for constraint solving: constraint propagation by domain reduction and (tree) search.

Constraint Propagation

The purpose of constraint propagation is to remove from the variable domains the values that do not contribute to any solution. For example if two integer variables $x, y \in [0..10]$ are constrained by $x < y$, we may remove 10 from the domain of x, and 0 from the domain of y.

In DICE constraint propagation is implemented by repeatedly applying a number of *domain reduction functions* (DRF's) until none of these can reduce the domains any further. The DRF's apply to variable domains, and enforce the constraints. They can be seen as incomplete solvers, that reduce a CSP to a simpler problem, while maintaining the set of solutions.

Search

In general constraint propagation will not be able to solve a CSP, so eventually, constraint solving comes down to searching. At each node of the search tree, starting with the node that represents the original problem, DICE applies constraint propagation before search progresses, in order to prune the remaining search space. Search is further characterized by two aspects: a *splitting strategy*, which expands the search tree, and a *search strategy*, which determines how to traverse the search tree.

When constraint propagation finishes in a node of the search tree, one of three situations occurs. If the domain of one or more variables has become empty, the node represents a failure. If all domains have been reduced to singleton sets, the node represents a solution. In all other cases, the node is an internal node of the search tree, and the splitting strategy is consulted to select a variable for which there are still alternative possible assignments. This aspect of splitting is referred to as *variable selection*. An example variable selection strategy is *fail-first*, which selects a variable for which the number of remaining alternatives is minimal. Then a number of sub-CSP's are created by actually splitting the domain of this variable according to a particular *value selection* method. Example value selection methods are bisection, which splits the domain in two halves, and enumeration, which generates a sub-CSP for every value in the original domain. Together the solutions to these sub-CSP's constitute the set of solutions to the original CSP. Every sub-CSP created according to the splitting strategy is a new node of the search tree, where constraint propagation can be applied again. The

search strategy determines the order in which the nodes of the search tree are visited by propagation and subsequent splitting.

2.2 Coordination Model and Language

Coordination languages offer language support for composing and controlling software architectures made of parallel or distributed components [13]. In the IWIM model of coordination [4], 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 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 [4, 5] 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.

3. Coordination Protocols

3.1 A Distributed Constraint Propagation Algorithm

The purpose of a constraint propagation algorithm in this context 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, DICE implements the coordination-based chaotic iteration algorithm of [11]. 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 variable 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 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

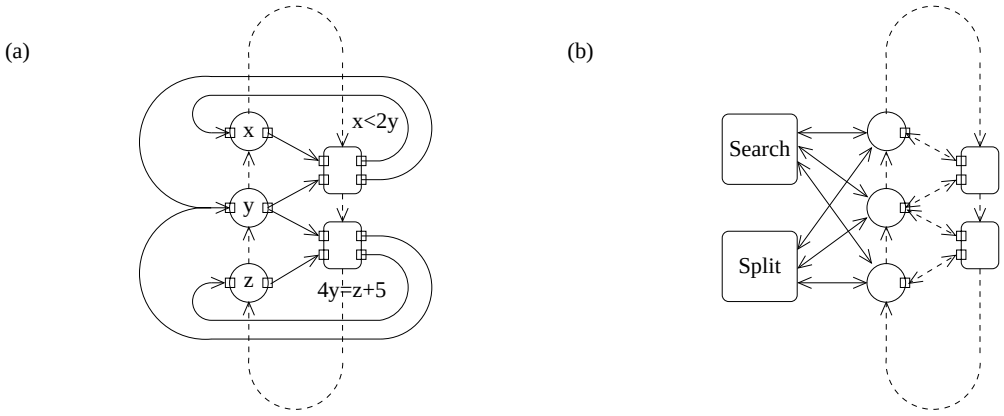

Figure 1: (a) An example process network of the distributed constraint propagation algorithm, (b) the propagation engine is coordinated from the outside to perform search

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 [11] it is argued that this distributed algorithm implements a restriction of the Generic Iteration Algorithm for Compound Domains of [1]. 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 DRF's are inflationary. The latter is effectively ensured by having Variable processes compute intersections.

3.2 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 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 [6] to detect these conditions. For the purpose of this algorithm, the processes of the constraint propagation algorithm are connected in a ring network. The

[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.

dashed lines in Figure 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.

3.3 Distributed Splitting

DICE employs a scheme similar to that of [3], 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::		*DRF*::	
v:	$World \xrightarrow{m} Domain$	I:	ARRAY $[1..n]$ OF $World \xrightarrow{m} Domain$
color:	$World \xrightarrow{m} \{\text{black}, \text{white}\}$	*color*:	$World \xrightarrow{m} \{\text{black}, \text{white}\}$
n_msg:	$World \xrightarrow{m} \mathbb{Z}$	*n_msg*:	$World \xrightarrow{m} \mathbb{Z}$

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. Maps v and I hold the data for the propagation algorithm, and *color* and *n_msg* represent the state of the termination detection algorithm.

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*, which implement the splitting strategy (involving variable selection and value selection) and search strategy, respectively. These processes have connections to all Variable processes (Figure 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 split (if any), and construct a set of new *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 new worlds have become available.

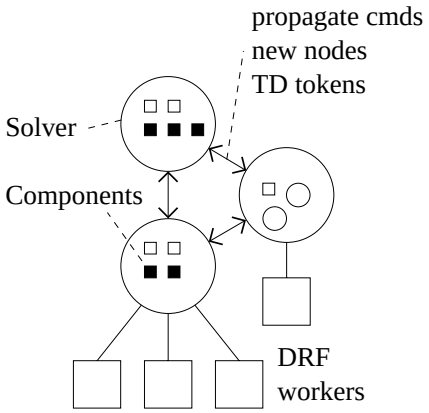

Figure 2: Example DICE network

4. Cooperating Solvers

The design of Section 3 supports the cooperation of solvers on the level of reduction steps inside the branch-and-prune 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 we adopt a more general scheme, which is comparable to that of [12], 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 DRF's to variables that it maintains,
- split 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 4.1 and 4.2 we discuss some details and implications of this scheme, related to constraint propagation and search, respectively. In Section 4.3 we introduce DRF worker processes to support parallel search. Figure 2 shows an example network of solvers.

4.1 Grouping Variables and Reduction Operators

Solver processes can have a pair (input and output) of ports for any variable 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 fixpoint of their local DRF's. 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 3.1. Incoming propagate commands that reduce the domain of a CSP variable trigger the fixpoint computation.

Compared to the design of Section 3.1, 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 only takes place for CSP variables that are shared by solvers. Based on the results reported in [12], we can expect that for sufficiently large

problems, a partitioning of CSP variables and DRF's can be found for which efficiency can be gained from distributed execution.

The DRF processes of Section 3.1 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 DRF's. 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 to solvers that are interested in this information. 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 provide a default topology that ensures the maximum level of consistency that can be achieved by the domain reduction functions.

4.2 Search by Cooperating Solvers

More than one splitting 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 split 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.

For complementary strategies, facilities will be provided that allow cooperating solvers to adhere to a common variable selection method. An example is discussed in Section 5.1. 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 splitting 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 agent of Section 3.3. Reports of new nodes created by another solver are treated by the receiving solver in the same way as new nodes created internally. Because a node is labeled with the solver process and splitting 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 search strategy components, 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 will be running 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 3.3, where search 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 search (traversal) strategy.

4.3 Parallel Search by Delegation

As a second extension to the design of Section 3, solver processes in DICE are allowed to delegate the actual application of domain reduction functions and splitting 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 purpose 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 compute-intensive reduction steps. The primary example would be a full solver, that autonomously explores part of the search tree, and splits the node of the search tree into a set of subtrees 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 fixpoint of the DRF's 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 have chosen 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. Also note that for solver cooperation schemes aimed at exploiting parallelism to achieve a reduction of turn-around time, load balancing and adapting to CPU availability become important. These are good examples of opportunities for developing or reusing independent coordinator modules.

In the model of constraint solving described in Section 2.1 search is defined as the scheduling of nodes of the search tree that await constraint propagation and subsequent splitting. It should be noted that compared to the Asynchronous Backtracking algorithm, and derived algorithms for solving distributed CSP's described in [14], DICE focusses on distributed constraint propagation instead of distributed search. Different solver processes may be running different search strategies concurrently, and several autonomous solvers may explore different parts of the search space in parallel, but the scheduling of nodes for propagation and splitting is a synchronous and sequential operation. On each path of the search tree variables are instantiated one after the other.

5. Implementation

The functionality of the DICE processes is left unspecified to a large extent. These processes are empty shells, executing the coordination protocols discussed in the previous sections, that accommodate software components that perform the actual constraint solving. The solver processes provide the

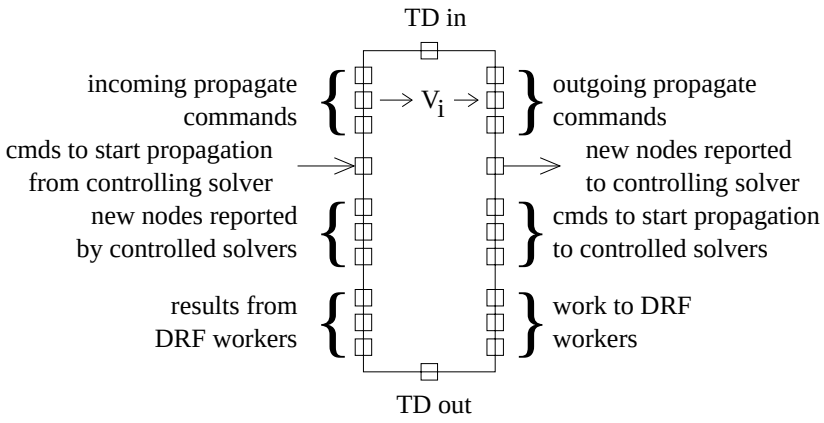

Figure 3: Port footprint of a solver process

protocols as coordination services to the components. Figure 3 shows the ports of a solver process used to implement these coordination protocols. In this section we discuss setting up a DICE network from user (5.1) and software engineering (5.2) points of view. The current implementation is discussed in Section 5.3.

5.1 Configuration File

To facilitate experiments, the initial DICE process reads a configuration file, and sets up the network of solver and DRF worker processes accordingly. Figure 4 shows a (contrived) example of a DICE configuration file.

```
SOLVER SolverXY(y) PARALLEL 2
{
   VARIABLE x IS DiscreteDomain { 0..10 }
   VARIABLE y IS DiscreteDomain { 0..10 }

   DRF IntegerArithmetic { x < 2*y }
   DRF GlobalFailFirst { x,y }
}
SOLVER SolverYZ(y)
{
   VARIABLE y;
   VARIABLE z IS DiscreteDomain { 0..10 }

   DRF IntegerArithmetic { 4*y = z+5 }
   DRF GlobalFailFirst { z }
}
SOLVER SearchAgent CONTROLS SolverXY,SolverYZ
{
   SEARCH DepthFirst { 2 }
}
```

Figure 4: A DICE configuration file

In this example, three solvers are employed to find the integer solutions to the system $x, y, z \in [0..10]$, $x < 2y$, $4y = z + 5$. The first solver maintains variables x and y, and the second solver maintains variables y and z. Both solvers export y, and exchange information about this variable by means

⟨*Cooperation*⟩ → ⟨*SolverSpec*⟩ {⟨*SolverSpec*⟩}

⟨*SolverSpec*⟩ → ⟨*SolverHeader*⟩ ;
→ ⟨*SolverHeader*⟩ ⟨*Specifier*⟩

⟨*SolverHeader*⟩ → `SOLVER` ⟨*Identifier*⟩
[`(` ⟨*IdentifierList*⟩ `)`]
[`CONTROLS`⟨*IdentifierList*⟩]
[`PARALLEL`⟨*Integer*⟩]

Figure 5: Solver cooperation specification language

of the distributed constraint propagation algorithm. x and z are internal variables. By means of a `GlobalFailFirst` component, solver `SolverXY` is allowed to split variables x and y in order to generate new nodes of the search tree. Solver `SolverYZ` can split variable z.

`GlobalFailFirst` splits a variable in the solver that it has been installed into, according to the default value selection method for that variable's domain type, if that variable has the smallest domain size (greater than one) in the entire network. Information about the domain sizes is collected by the tokens of the termination detection algorithm, as a special service that allows cooperating splitting strategies to adhere to a global variable selection method.

A third solver `SearchAgent` gathers the information about new nodes in the search tree created by the other two solvers. A component `DepthFirst` has been installed in this solver, that tries to keep propagation running in two nodes simultaneously. The set of nodes of the search tree that await propagation is managed by the `DepthFirst` component as a stack. Further notable aspects of this example are:

- `SolverXY` has two DRF workers to which it delegates the actual execution of its domain reduction function and splitting strategy. `SolverYZ` has no DRF workers, and is running these components itself. `SearchAgent` does not have any DRF or splitting strategy components installed, so no work can be delegated by it.
- For the purpose of the constraint propagation algorithm, the solvers are connected according to the default network topology. The default is that the first solver that exports a particular variable will become the *anchor* for that variable. All other solvers that export the same variable are connected to it. The anchor plays the role of the Variable processes of Section 3.1.
- Constraint propagation starts as soon as the first domain reduction function is installed, but splitting does not start until termination of constraint propagation is detected. The termination detection algorithm is started by installing a search strategy component.
- Additional statements exist for gathering and printing solutions.

5.2 Component Model

The parsing of configuration files happens at three levels:

1. The initial DICE executable interprets a sequence of solver specifications, according to a simple language, part of whose syntax is shown in Figure 5, where {...} should be read as "followed by zero or more instances of the enclosed," and [...] indicates an optional syntax element. Solver names, exported variables and number of DRF workers are interpreted, but solver *specifiers* are sent to solver processes. In the example of Figure 4, this is all the text inside the outer curly braces. This text is meaningless from the point of view of the top level parser.

⟨*Solver*⟩ → ⟨*Statement*⟩ {⟨*Statement*⟩}

⟨*Statement*⟩ → `VARIABLE` ⟨*Identifier*⟩
`IS` ⟨*Identifier*⟩ ⟨*Specifier*⟩
→ `DRF` ⟨*Identifier*⟩ ⟨*Specifier*⟩
→ `SEARCH` ⟨*Identifier*⟩ ⟨*Specifier*⟩

⟨*Specifier*⟩ → {⟨*String*⟩}

Figure 6: Solver configuration language syntax

2. Solvers parse specifier strings, sent to them by the initial DICE executable just after process creation, as sequences of component specifications.[2] Part of the language for this is shown in Figure 6. A component specification consists of an identifier and a specifier (and in the case of variables, an extra identifier for the variable name). The identifier designates a particular component. The specifier is used to create an instance of this component. This instance is installed in the solver. Again, the specifier strings have no meaning to the solver process. These are parsed by the components. Examples of these are the texts inside the inner curly braces of the example in Figure 4.

3. Different components parse different languages. For example, the `DiscreteDomain` component, which is in the "variable domain type" category, parses string like `1..10`, and `IntegerArithmetic`, in the domain reduction function category, reads strings that represent relations between pairs of polynomials.

Programming a DICE component involves writing a C++ subclass of one of three abstract classes, corresponding to the rules for *Statement* in Figure 6, that implement the four component categories:

- class `Domain` for implementations of domains of CSP variables,
- class `DRF_Component` for DRF's and splitting strategy implementations,
- class `SearchComponent` for search strategy implementations.

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, VarSelInfo *vsinfo,
                            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:

[2]Because Manifold processes may be running as threads in the same address space, the parsers at different levels have to be thread-safe. For this purpose we use the `flex++` and `bison` tools to generate the main DICE parser, the solver parsers, and the parsers for several of the more complicated components.

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 `DescriptionToDRF_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].

3. By means of the virtual member functions `Compute` and `Termination`, a `DRF_Component` can act as a domain reduction operator, and react when termination is detected, respectively. Regular DRF components implement only the `Compute` member function, and splitting strategies implement only `Termination`. The parameters to `Compute` include an array of `Domain` pointers, corresponding to the parameters of the domain reduction function. Additionally, `Termination` has parameters for identifying the nature of the node of the search tree for which termination is detected (solution, failure, or internal), and to pass the information about the variables that is collected for the purpose of implementing a global variable selection method. `Domain` offers facilities for adding subdomains to a variable domain. These are transformed by the framework into new worlds. In Section 5.3 the `Minimize` component is discussed. This is an example of a component that implements both `Compute` and `Termination`.

4. In order for a solver process to be able to relate the buffers associated with its input ports to the arguments of its domain reductions function, `DRF_Component::VariableNames` should supply a vector of variable names. The elements of this vector must 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 of operators on small finite domains [2].

The solver and DRF worker processes execute a command loop. All coordination protocols are implemented by these processes exchanging commands. Such commands are Manifold *tuples*. As an example, the following command implements the propagate command of the distributed constraint propagation algorithm:

$<$`CMD_SOLVER_PROPAGATE`$, w, d >$

Here w (world) identifies the node of the search tree that the command applies to, and d is the unit representation of the new domain of the CSP variable in that node. The handler for `CMD_SOLVER_PROPAGATE` calls a member function `Update` on the variable's current domain in world w, using the domain represented by d as an argument.

An example of a (non-distributed) system that is open-ended in a similar way is Figaro [9]. Also in Figaro 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.

[3] At a later stage we may want to switch from these static functions to a plug-in system, using dynamic loading to take component code on board.

5.3 Status

The current implementation of the DICE framework still closely follows the design of Section 3. There exist four kinds of processes, Variable, DRF, Search, and Split, that act as placeholders for software components in the corresponding categories. The initial process interprets a language similar to that of Figure 6 to set-up the process network. The facilities described in Section 4, and the extra level of parsing of Figure 5 are being implemented. This implementation will focus on speedup as a result of parallel search.

For this purpose, as suggested in Section 4.3, we plan to implement a splitting strategy component that acts as a full sequential solver, but stops searching after a certain time-out has elapsed. At that time, it dumps all solutions, and a (preferably minimal) number of nodes for the part of the search space that still needs to be explored back into the (network of the) solver cooperation. This mechanism can be used for implicit load balancing.

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. `DoubleInterval` provides a representation of real numbers by intervals, of which the bounds are double-precision floating point numbers.

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 the JAIL library [7].

`IntegerPairs` is initialized by two variable names, and a set of integer pairs that constrains the values that these variables may assume. For example[4]:

```
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 a specifier 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. 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. It is also possible to group several variables into a single process. These facilities will be obsolete with the introduction of solver processes, as discussed in Section 4.1.

As an alternative to having special components that implement optimizing search strategies, such as branch-and-bound, a `Minimize` component is available, which can be used to transform any search

[4] The configuration language of the current implementation uses quotes to delimit specifier strings. The `ENUM` keyword provides facilities for using symbolic names for integer constants. See [15] for details.

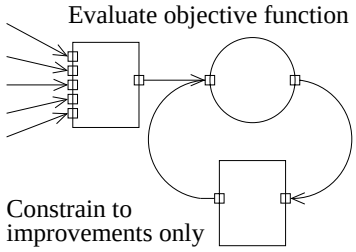

Figure 7: The approach to optimization encouraged by DICE

strategy into an optimization strategy. Figure 7 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` enforces a dynamic constraint, stating that only improvements of this bound will be considered as new solutions.

Splitting Strategy

Currently, only one splitting strategy implementation exists, which splits a variable domain according to the default splitting strategy 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 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 demonstrates the effectiveness of DICE on many standard examples. However, much work still needs to be done before we can claim to have a general-purpose constraint solver. Notably, our integer arithmetic presently cannot multiply variables, our constraints over the reals are currently limited to quadratic equations, our default splitting strategy 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 that we can use DICE as a platform for experimenting with

different search procedures, alternatives must be provided in the splitting strategy and 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.

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 strategy that dynamically changes from one basic splitting strategy 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. Applications

The current version of the framework, described in Section 5.3, demonstrates the effectiveness of the coordination-based approach, but does not yield efficient solvers. The primary reason is that communication is involved with every application of a DRF, and every node of the search tree. On top of this, the termination detection algorithm that we use has a considerable overhead, and a DICE solver is easily outperformed by any sequential solver on standard combinatorial benchmarks like the n-queens problem.

However, the current implementation is well suited for handling distributed CSP's [14], where knowledge of the constraints is distributed, while it is impractical or impossible to gather this information in a single solver. Imagine for example that constraints are defined by the contents of databases, running on different machines. DRF components could be written that query these databases to derive and enforce the constraints.

In the remainder of this section we discuss a number of possible applications of the facilities introduced in Section 4.

Fully Distributed Solver

The current implementation can still be emulated by having a separate solver for every variable, for every domain reduction function and for the splitting strategy. Also one solver will be dedicated to collecting the new nodes of the search tree that are generated by splitting. The solvers corresponding to the variables serve only to forward propagate commands that actually reduced the domain of a CSP variable.

Slightly more communication is involved in the new design, because originally, Variable processes had separate ports for communicating with the Split process. Now the solver that implements the splitting strategy is notified of every domain reduction, while it will only act on termination of constraint propagation.

Sequential Solver

At the other end of the spectrum, all variables and domain reduction functions can be loaded into a single solver that does not have to participate in the distributed termination detection algorithm, and need not examine its external ports for new information generated by other processes. Splitting, and scheduling the nodes of the search tree for application of the domain reduction functions are done internally. Several such sequential solvers can be running concurrently, working on the same problem, but using different strategies in an attempt to find a first solution quickly.

Parallel Search

A full solver can be incorporated into the framework as a splitting strategy component that generates a new node for every solution that it finds. These leaves of the search tree are direct descendants of the internal node that represents the problem that the full solver was working on. The DRF workers can be used to have several such full solvers exploring different parts of the search tree in parallel.

One option would be to split in the conventional way up to a certain depth of the search tree, and to apply a full solver once this depth has been reached. This threshold depth should be chosen such that a sufficiently large amount of subproblems is available for parallel exploration. Alternatively, the "time-out sequential search" component, described in Section 5.3, can be used for implicit load balancing.

Parallel Branch-and-Bound

Together with the `Minimize` component discussed in Section 5.3, the previous cooperation scheme using the time-out sequential search component implements parallel optimization. After the time-out value, any part of the search space that still needs to be explored is first taken through the global propagation mechanism, where `Minimize` may reduce the domain of the variable that implements the objective function. Depending on the approximation of this objective function used in internal nodes of the search tree, this may prevent the sequential solvers from exploring subtrees that cannot improve the bound.

For optimization, it is important that solutions (singleton sets) are propagated first, in order that the bound can be updated as soon as possible. Also if the time-out sequential search component is allowed to dump singleton sets that have not been subject to propagation, the global solver should be loaded with enough reduction operators to filter out the solutions.

Symbolic Solvers

Currently, cooperating solvers in DICE are constructed from components that are coded inside the framework, but the IWIM design and Manifold implementation enable us to interface with existing solvers as well. A particular solver cooperation scheme that we want to implement is to have a symbolic solver compute redundant equations in a side process, to speed-up the search for solutions to a system of non-linear equations [8]. The current implementation is suited for this purpose because it allows the DRF processes that enforce these extra constraints to be added to a live network in the middle of the solving process.

However, because our current termination detection algorithm requires correct counters of the number of messages in the network this involves a large penalty. To make a Variable to DRF channel connection, we must (1) stop the Variable process, (2) make the connection, (3) update the counter of outgoing channels for that Variable, and (4) start the Variable again. This way, a Variable knows exactly how many duplicates are generated of each reduction request. We can avoid this overhead by using a more sophisticated termination detection algorithm, but in the design of Section 4 we also have the option to let the external solver send the specifications for the new domain reduction functions, coded in the language of Figure 6, to an existing solver process.

Local Search to Speed up Optimization

Similarly, a local search solver can generate constraints that reduce the domain of the variable holding the outcome of the objective function in an optimization problem. An example application would be a graph coloring problem, where branch-and-bound is used to ensure that a minimal coloring is found. The first solution found by the local search solver will limit the domains of the color variables.

In this case it would be desirable to allow the external solver to produce a continuous stream of (suboptimal) solutions, that hopefully contain further improvements of the bound. Extension of the framework with coordination protocols for starting external solvers, directing their output to specific solver processes, and for stopping them when the global search has finished will be straightforward.

7. Conclusion

In this report we have given an overview of the DICE framework, and discussed our plans for developing it further. DICE is distinguished from other platforms for constraint solving by the combination of an open-ended and inherently distributed design. The heart of the framework is the coordination-based chaotic iteration algorithm introduced in [11]. The proposed extensions can be seen as optimizations: several components from the original design can be grouped into a single process to reduce the communication volume, and the delegation of work to DRF worker processes increases the capacity of the propagation engine in order that the nodes of the search tree that are being processed simultaneously can actually be processed in parallel.

The main type of process in the target system is a solver. These processes can accommodate any combination of variables, domain reduction functions, splitting strategies and search strategies. Depending on the functionality thus loaded into the solver processes, the resulting distributed system can be seen as a single distributed constraint solver, or as a number of individual constraint solvers that cooperate to solve a problem. Reasons for distributing solver components include:

- constraint satisfaction problems of a distributed nature, where not performance, but the physical distribution of the constraints is the main issue, and
- the expectation that a distributed system can solve a problem more efficiently, for example by exploiting parallelism, or by having several solvers, using different solving techniques, cooperate to solve a problem.

In Section 6 we discussed several examples of how DICE can be used as a platform for solver cooperation, but compared to a solver cooperation language such as BALI (see for example [13]), and the system described in [10] the possibilities are limited: the cooperation scheme is always the distributed fixpoint computation, and the model of constraint solving is fixed to branch-and-prune search. However, in [11] it is explained how the distributed propagation algorithm can be forced to apply DRF's in a specific sequence. Also because DICE abstracts from variable domain types, the propagation algorithm could be used to transform problems, instead of domains of CSP variables. This would bypass the search mechanism, and allow for a large variety of models of constraint solving. Using DICE, or in general, chaotic iteration for this kind of solver cooperation is a subject for future research.

Acknowledgment

I am much indebted to Farhad Arbab, Krzysztof Apt and Eric Monfroy for discussions and careful reading of the material presented here.

References

1. K.R. Apt. The Rough Guide to Constraint Propagation. In J. Jaffar (ed.) *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, LNCS 1713, pp. 1–23, Springer-Verlag, 1999. Invited lecture.
2. K.R. Apt, E. Monfroy. Automatic Generation of Constraint Propagation Algorithms for Small Finite Domains. In J. Jaffar (ed.) *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, LNCS 1713, pp. 58–72, Springer-Verlag, 1999.
3. 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.
4. 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.
5. 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.
6. E.W. Dijkstra. *Shmuel Safra's Version of Termination Detection* (Note EWD998).
7. F. Goualard. *JAIL - Just Another Interval Library*, Manual rev. 0.0.9, Institut de Recherche en Informatique de Nantes, January 2002.
8. L. Ganvilliers, E. Monfroy, and F. Benhamou. Symbolic-Interval Cooperation in Constraint Programming. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation, ISSAC'2001*, pages 150–166. ACM Press, 2001.
9. 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.
10. P. Hofstedt. Better Communication for Tighter Cooperation. In Lloyd et al. (eds.) *Proceedings of the 1st International Conference on Computational Logic (CL 2000)*, LNAI 1861, pp. 342–357, Springer-Verlag, 2000.
11. 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.
12. E. Monfroy and J.-H. Réty. Chaotic Iteration for Distributed Constraint Propagation. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 19–24. ACM Press, 1999.

13. 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.
14. M. Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*, Springer-Verlag, 2001.
15. P. Zoeteweij. *A Coordination-Based Framework for Parallel Constraint Solving*. Presented at the ERCIM/CologNet workshop on Constraint Solving and Constraint Logic Programming, Cork, 2002. Available at http://www.cs.ucc.ie/∼osullb/ercim2002.
16. P. Zoeteweij. *Coordination-Based Solver Cooperation in DICE*. Presented at the COSOLV'2002 Workshop on Cooperative Solvers in Constraint Programming (held in conjunction with CP2002), Ithaca, NY, USA, 2002.
17. P. Zoeteweij. Coordination-Based Distributed Constraint Solving in DICE. In *Proceedings of the 2003 ACM Symposium on Applied Computing*. To appear.