

```
R={New Remote.manager init}
```

creates a remote process and a local object `R` which is the remote process's "manager". If no arguments are given, then the remote process is created on the same machine as the caller process. With the right arguments it is possible to create processes on other machines. For example, the call:

```
R={New Remote.manager init(host:'norge.info.ucl.ac.be')}
```

creates a remote process on the host `norge.info.ucl.ac.be`. By default, the remote process will be created using `rsh` (remote shell). In order for this to work, the host must have been set up properly beforehand. The remote process can also be created with `ssh` (secure shell). For information on this and other aspects of `Remote`, please see the Mozart documentation [129].

Once a remote process has been created, it can be controlled through the manager object `R`. This object has an interface that closely resembles that of `Module`, i.e., it controls the instantiation of functors at the remote process. Calling the manager with `{R apply(F X)}` installs functor `F` on the remote process and returns the module in `X`.

There is a kind of "master-slave" relationship between the original process and the new process. The original process can observe the new process's behavior, for example, to keep track of its resource consumption. The original process can change the new process's process priority, put it to sleep, and even terminate it if necessary. The original process can give the new process limited versions of some critical system modules, so that the new process behaves like a sand box.

11.7 Distribution protocols

We now briefly summarize the distribution protocols implemented in Mozart. We first give an overview of all the different protocols for the different language entities. We then focus on two particularly interesting protocols: the mobile state protocol (used for cached cells and objects) and the distributed binding protocol (used for dataflow variables). We end with a quick look at the distributed garbage collector.

11.7.1 Language entities

Each language entity is implemented by one or more distributed algorithms. Each algorithm respects the entity's semantics if distribution is disregarded. The language entities have the following protocols:

- Stateful entities are implemented with one of the following three protocols:
 - **Stationary state.** All operations always execute on the process where the state resides, called the target process. Remote invocations send messages to this process. Conceptually, it is as if the invoking

thread moves to the target process. When seen in this way, distributed exceptions and reentrant locking work correctly. Operations are synchronous. Asynchronous operations require explicit programming, e.g., by using **thread ... end**.

- **Mobile state.** In this case the invoking thread is stationary. The right to update the state moves from one process to another. We call this right the *state pointer* or *content edge*. An exchange operation will first cause the state pointer to move to the executing process, so that the exchange is always local [201, 197]. The mobile state protocol can be seen as implementing a cache, i.e., it is a *cache coherence* protocol.
- **Invalidation.** This protocol optimizes the mobile state protocol when reading is much more frequent than updating. A process that wants to read the state sends a message to the target process and gets the state in reply, thus creating a local replica of the state. A process that wants to update the state must first explicitly *invalidate* all these replicas by sending them an invalidation message. This guarantees that the interleaving semantics of state updates is maintained. The right to update the state still moves from one process to another.
- Single-assignment entities are implemented with a distributed unification algorithm [71]. The key operation of this algorithm is a distributed bind operation, which replaces the variable by whatever it is bound to, on all the processes that know the variable. There are two variants of this algorithm:
 - **Lazy binding** (on demand). The replacement is done on a process only when the process needs the variable. This variant reduces the number of network messages, but increases latency and keeps a dependency on the variable's home process.
 - **Eager binding** (on supply). The replacement is done on a process as soon as the variable is bound, whether or not the process needs the variable.

The Mozart system currently implements just the eager binding algorithm.

- Stateless entities are implemented with one of the following three protocols [3]:
 - **Lazy copying** (on demand). The value is copied to a process only when the process needs it. This reduces the number of network messages, but increases latency and keeps a dependency on the original process.
 - **Eager copying** (on supply, sent if not present). The value is not sent as part of the message, but if upon reception of the message the value is not present, then an immediate request is made for it. In most

Kind of entity	Algorithm	Entity
Stateless	Eager immediate copying	record, integer
	Eager copying	procedure, class, functor
	Lazy copying	object-record
Single assignment	Eager binding	dataflow variable, stream
Stateful	Stationary state	port, thread, object-state
	Mobile state	cell, object-state

Table 11.1: Distributed algorithms

cases, this is the optimal protocol, since the value will be present on the receiving process for all except the first message referencing it.

- **Eager immediate copying** (on supply, always sent). The value is sent as part of the message. This has minimum latency, but can overload the network since values will be repeatedly sent to processes. It is used to send record structures.
- In addition to these algorithms, there is a distributed garbage collection algorithm. This algorithm works alongside the local garbage collection. The algorithm does no global operations and is able to remove all garbage except for cross-process cycles between stateful entities. The algorithm consists of two parts: a credit mechanism and a time-lease mechanism. The credit mechanism works well when there are no failures. It is a kind of weighted reference counting [151]. Each language entity with remote references has a supply of “credits”. Each remote reference to the entity must have at least one credit. When a local garbage collection reclaims a remote reference, then its credits are sent back. When the entity has no outstanding remote credits and no local references, then it can be reclaimed. In the time-lease mechanism, each distributed entity exists only for a limited time unless it is periodically renewed by a message sent from a remote reference. This handles the case of partial failure.

Table 11.1 shows the algorithms used by the current system for each language entity. In this table, an object consists of two parts, the *object-record* (which contains the class) and the *object-state* (the object’s internal cell). We conclude that network operations⁶ are predictable for all language entities, which gives the programmer the ability to manage network communications.

11.7.2 Mobile state protocol

The mobile state protocol is one of the distributed algorithms used to implement stateful entities. Objects, cells, and locks are implemented with this protocol. This section gives the intuition underlying the protocol and explains the network

⁶In terms of the number of network hops.

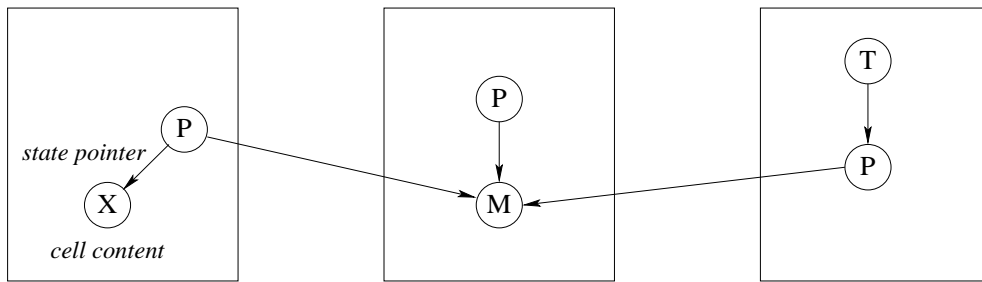


Figure 11.6: Graph notation for a distributed cell

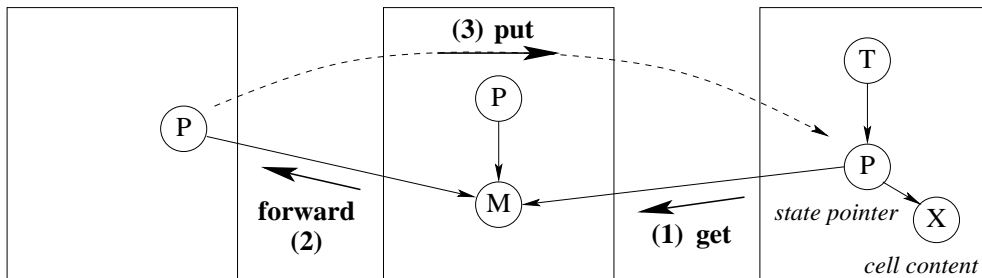


Figure 11.7: Moving the state pointer

operations it does. A formal definition of the protocol and a proof that it respects the language semantics are given in [201]. An extension that is well-behaved in case of network and process failure is given together with proof in [20]. The Mozart system implements this extended protocol.

We use a *graph notation* to describe the protocol. Each (centralized) language entity, i.e., record, procedure value, dataflow variable, thread, and cell, is represented by a node in the graph. To represent a distributed computation, we add two additional nodes, called proxy and manager. Each language entity that has remote references is represented by a star-like structure, with one manager and a set of proxies. The proxy is the local reference of a remote entity. The manager coordinates the protocol that implements the distribution behavior of the entity. The manager is also called the coordinator.

Figure 11.6 shows a cell that has remote references on three processes. The cell consists of three proxies *P* and one manager *M*. The cell content *X* is accessible from the first proxy through the state pointer. A thread *T* on the third process references the cell, which means that it references the third proxy.

What happens when *T* does an exchange operation? The state pointer is on a different process from *T*, so the mobile state protocol is initiated to bring the state pointer to *T*'s process. Once the state pointer is local to *T*, then the exchange is performed. This implies the remarkable property that all cell operations are always performed locally in the thread that initiates them.

The protocol to move the state pointer consists of three messages: **get**, **put**,

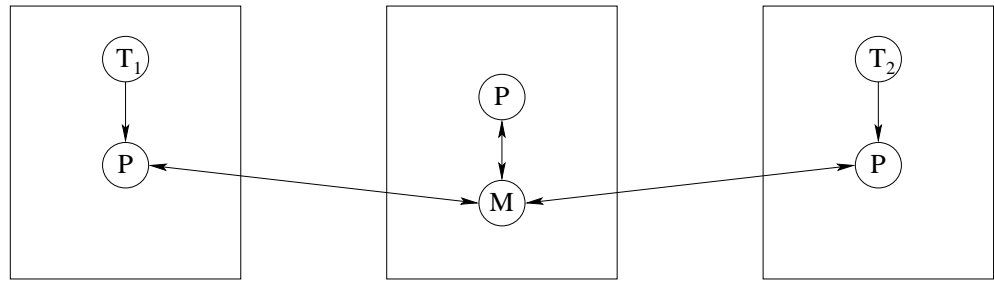


Figure 11.8: Graph notation for a distributed dataflow variable

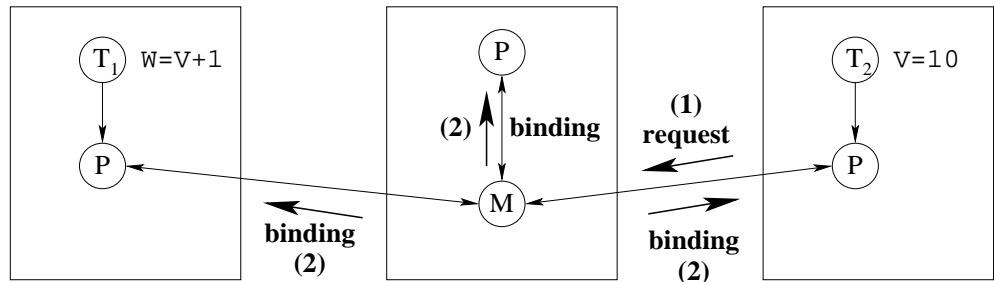


Figure 11.9: Binding a distributed dataflow variable

and **forward**. Figure 11.7 shows how they are sent. The third proxy initiates the move by sending a **get** request to M. The manager M plays the role of a serializer: all requests pass through it. After receiving the **get**, M sends a **forward** message to the first proxy. When the first proxy receives the **forward**, it sends a **put** to the third proxy. This atomically transfers the state pointer from the first to the third proxy.

11.7.3 Distributed binding protocol

The distributed binding protocol is used to bind dataflow variables that have references on several processes. The general binding algorithm is called *unification*; the distributed version does distributed unification. A formal definition of the protocol and a proof that it respects the language semantics are given in [71]. The Mozart system implements an extended version of this protocol that is well-behaved in case of network and process failure.

When unification is made distributed it turns out that the whole algorithm remains centralized except for one operation, namely binding a variable. To give the intuition underlying distributed unification it is therefore sufficient to explain distributed binding.

Figure 11.8 shows a dataflow variable V that has remote references on three processes. Like the distributed cell, there are three proxies P and one manager M . The manager has references to all proxies. On the first process, thread T_1 references V and is suspended on the operation $w=v+1$. On the third process,

thread T_2 also references V and is about to do the binding $v=10$.

The protocol to bind V consists of two messages: `request(X)` and `binding(X)`. Figure 11.9 shows how they are sent. The third proxy initiates the protocol by sending `request(10)` to M . The first such request received by M causes a `binding(10)` to be sent to all proxies. This action is the heart of the algorithm. The rest is details to make it work in all cases. If M is already bound when it receives the request, then it simply ignores the request. This is correct since the proxy that sent the request will receive a binding in due course. If a new proxy is created on a fourth process, then it must register itself with the manager. There are a few more such cases; they are all explained in [71].

This algorithm has several good properties. In the common case where the variable is on just two processes, for example where the binding is used to return the result of a computation, the algorithm's latency is a single round trip. This is the same as explicit message passing. A binding conflict (an attempt to bind the same variable to two incompatible values) will cause an exception to be raised on the process that is responsible for the conflict.

11.7.4 Memory management

When distribution is taken into account, the Mozart system has three levels of garbage collection:

- A local garbage collector per process. This collector coordinates its work with the distributed collectors.
- A distributed garbage collector that uses weighted reference counting.
- A distributed garbage collector based on a time-lease mechanism.

Weighted reference counting The first level of distributed garbage collection uses weighted reference counting [151]. This collector works when there are no failures. It can rapidly remove all distributed garbage except for cross-process cycles between stateful entities on different owner processes. Each remote reference has a nonzero amount of credit, that implies the continued aliveness of the entity. When the remote reference is reclaimed, the credit is returned to the owner. When the owner sees there is no longer any outstanding credit, then the entity can be reclaimed if there are no more local references.

Weighted reference counting is efficient and scalable. First, creating a new remote reference requires essentially zero network messages in addition to the messages sent by the application. Second, each remote process does not need to know any other process except the owner process. Third, the owner process does not need to know any remote process.

Time-lease mechanism The second level of distributed garbage collection uses a time-lease mechanism. This collector works when there are permanent or long-lived temporary failures. Each remote reference has only a limited lifetime (a “lease on life”), and must periodically send a “lease renewal” message to the owner process. If the owner process does not receive any lease renewal messages after a given (long) time period, then it assumes that the reference may be considered dead.

The time-lease mechanism is complementary to weighted reference counting. The latter reclaims garbage rapidly in the case when there are no failures. The former is much slower, but it is guaranteed in finite time to reclaim all garbage created because of failure. This plugs the memory leaks due to failure.

Programmer responsibility The main problem with distributed garbage collection is to collect cycles of data that exist on several processes and that contain stateful entities. As far as we know, there does not exist an efficient and simple distributed garbage collector that can collect these cycles.

This means that distributed memory management is not completely automatic; the application has to do a little bit of work. For example, consider a client/server application. Each client has a reference to the server. Often, the server has references to the clients. This creates a cycle between each client and the server. If the client and server are on different processes, they cannot be reclaimed. To reclaim a client or a server, it is necessary to break the cycle. This has to be done by the application. There are two ways to do it: either the server has to remove the client reference or the client has to remove the server reference.

Creating a ticket with `{Connection.offerUnlimited x T}` makes `x` a permanent reference. That is, `x` is added to the root set and is never reclaimed. Once-only tickets, i.e., those created with `{Connection.offer x T}`, can only be taken once. As soon as they are taken, `x` is no longer a permanent reference and is potentially reclaimable again.

It is possible to use distribution to reduce the time needed by local garbage collection. With `Remote`, create a small process that runs the time-critical part of the application. Since the process is small, local garbage collection in it will be very fast.

11.8 Partial failure

Let us now extend the distribution model with support for partial failure. We first explain the kinds of failures we detect and how we detect them. Then we show some simple ways to use this detection in applications to handle partial failure.

11.8.1 Fault model

The *fault model* defines the kinds of failures that can occur in the system and how they are reflected in the language by a failure detection mechanism. We have designed a simple fault model that captures the most common kinds of failures on the Internet. The Mozart system can detect just two kinds of failures, permanent process failure and network inactivity:

- Permanent process failure is commonly known as fail-silent with failure detection. It is indicated by the failure mode `permFail`. That is, a process stops working instantaneously, does not communicate with other processes from that point onwards, and the stop can be detected from the outside. Permanent process failure cannot in general be detected on a WAN (e.g., the Internet), but only on a LAN.
- Network inactivity is a kind of temporary failure. It can be either temporary or permanent, but even when it is supposedly permanent, one could imagine the network being repaired. It is different from process failure because the network does not store any state. Network inactivity is indicated by the failure mode `tempFail`. The Mozart system assumes that it is always potentially temporary, i.e., it never times out by default.

These failures are reflected in the language in two ways, either synchronously or asynchronously:

- Synchronous (i.e., lazy) detection is done when attempting to do an operation on a language entity. If the entity is affected by a failure, then the operation is replaced by another, which is predefined by the program. For example, the operation can be replaced by a raised exception.
- Asynchronous (i.e., eager) detection is done independent of any operations. A program first posts a “watcher” on an entity, before any problems occur. Later, if the system detects a problem, then it enables the watcher, which executes a predefined operation in a new thread. Watchers use the well-known heart-beat mechanism for failure detection.

The two failure modes, detected either synchronously or asynchronously, are sufficient for writing fault-tolerant distributed applications. They are provided by Mozart’s primitive failure detection module `Fault`.

Network inactivity

The network inactivity failure mode allows the application to react quickly to temporary network problems. It is raised by the system as soon as a network problem is recognized. It is therefore fundamentally different from a time out. A *time out* happens when a part of the system that is waiting for an event abandons the wait. The default behavior of TCP is to give a time out after some minutes.

This duration has been chosen to be very long, approximating infinity from the viewpoint of the network connection. After the time out, one can be sure that the connection is no longer working.

The purpose of `tempFail` is to inform the application of network problems, not to mark the end of a connection. For example, if an application is connected to a server and if there are problems with the server, then the application would like to be informed quickly so that it can try connecting to another server. A `tempFail` failure mode can therefore be relatively frequent, much more frequent than a time out. In most cases, a `tempFail` fault state will eventually go away.

It is possible for a `tempFail` state to last forever. For example, if a user disconnects the network connection of a laptop machine, then only he or she knows whether the problem is permanent. The application cannot in general know this. The decision whether to continue waiting or to stop the wait can cut through all levels of abstraction to appear at the top level (i.e., the user). The application might then pop up a window to ask the user whether to continue waiting or not. The important thing is that the network layer does not make this decision; the application is completely free to decide or to let the user decide.

Where to do time outs

A surprisingly large number of existing systems (both programming platforms and applications) incorrectly handle prolonged network inactivity. When there is a prolonged network inactivity during an operation, they do a time out: they abort the waiting operation and invoke an error handling routine. Often, this abort is irrevocable: it is impossible to continue the operation. Many operating system utilities are of this type, e.g., `ftp` and `ssh`.

The mistake in this approach is that the decision to time out is made at the wrong level. For example, assume there is a time out in a lower layer, e.g., the transport layer (TCP protocol) of the network interface. This time out crosses all abstraction boundaries to appear directly at the top level, i.e., to the user. The user is informed in some way: the application stops, or at best a window is opened asking confirmation to abort the application. The user does not have the possibility to communicate back to the timed-out layer. This limits the flexibility of the system.

The right approach is not to time out by default but to let the application decide. The application might decide to wait indefinitely (avoiding an abort), to abort immediately without waiting, or to let the user decide what to do. This greatly increases the perceived quality of the system. For example, a hard-mounted resource in the NFS file system offers the first possibility. The Stop button in recent Web browsers offers the third possibility.

11.8.2 Simple cases of failure handling

We show how to handle two cases, namely disconnected operation and failure detection. We show how to use the `Fault` module in either case.

Disconnected operation

Assume that you are running part of an application locally on your machine through a dialup connection. These connections are not meant to last for very long times; they are made for conversations, which usually are short. There are many ways a connection can be broken. For example, you might want to hang up to make an urgent phone call, or you are connected in an airport and your calling card runs out of cash, or the phone company just drops your connection unexpectedly.

You would like your application to be impervious to this kind of fickleness. That is, you would like the application to wait patiently until you reconnect and then continue working as if nothing went wrong. In Mozart, this can be achieved by setting the default failure detection to detect only permanent process failures:

```
% Each process executes this on startup:
{Fault.defaultEnable [permFail] _}
```

This means that operations will only raise exceptions on permanent process failures; on network inactivity they will wait indefinitely until the network problem goes away.

Detecting a problem and taking action

On many computers, booting up is an infuriating experience. How many times have you turned on a laptop, only to wait several minutes because the operating system expects a network connection, and has to time out before going on? Mozart cannot fix your operating system, but it can make sure that your application will not have the same brainless behavior.

Assume you want to use a remote port. If the remote port has problems (intermittent or no access) then the application should be informed of this fact. This is easy to set up:

```
% Get a reference to the port:
X={Take tickfile}

% Signal as soon as a problem is detected:
{Fault.installWatcher X [tempFail permFail]
  proc {$ _ _}
    {Browse X#` has problems! Its use is discouraged.`}
  end _}
```

The procedure passed to `Fault.installWatcher` is called a *watcher*; it will be called in its own thread as soon as the system detects a problem. It's up to you to

```

fun {NewStat Class Init}
  Obj={New Class Init}
  P
in
  thread S in
    {NewPort S P}
    for M#X in S do
      try {Obj M} X=normal
      catch E then
        try X=exception(E)
        catch system(dp(...) ...) then
          skip /* client failure detected */
        end
      end
    end
  end
  proc {$ M}
  X in
    try {Send P M#X} catch system(dp(...) ...) then
      raise serverFailure end
    end
    case X of normal then skip
    [] exception(E) then raise E end end
  end
end

```

Figure 11.10: A resilient server

do what's necessary, e.g., set an internal flag to indicate that no communication will be done.

If the problem was `tempFail`, then it is possible that communication with `x` will be restored. If that happens, Mozart allows you to continue using `x` as if nothing wrong had happened.

11.8.3 A resilient server

We saw the `NewStat` operation for creating stationary objects. Let us show how to extend it to be resilient to client failure and at the same time protect the client against server failure. We use the exception-handling mechanism. Attempting to perform an operation on an entity that requires coordination with a remote failed process will raise an exception. We use this behavior to protect both the server and the client from failures. We protect the server by using a **try** statement:

```
try {Obj M} X=normal catch ... end
```

This protects the server against a client that shares a variable with the server (in this case `x`). We need a second **try** statement:

```
try X=exception(E) catch _ then skip end
```

since the statement `X=exception(E)` also binds `x`. We protect the client:

```
proc {$ M}
x in
  try {Send P M#X}
  catch _ then raise serverFailure end end
  case X of normal then skip
  [] exception(E) then raise E end end
end
```

The `try {Send P M#X} ... end` signals to the client that the server has failed. In general, any operation on a distributed entity has to be wrapped in a `try`. The complete definition of `NewStat` is given in Figure 11.10. Note that distribution faults show up as exceptions of the form `system(dp(...) ...)`.

If `tempFail` detection is enabled, the stationary server defined here will be slowed down if there are communication problems with the client, i.e., it will wait until `tempFail` is raised (for example when `try X=exception(E)` is executed). One way around this problem is to provide multiple server objects to allow serving multiple clients simultaneously.

11.8.4 Active fault tolerance

Applications sometimes need *active fault tolerance*, i.e., part of the application is replicated on several processes and a replication algorithm is used to keep the parts coherent with each other. Building ADTs to provide this is an active research topic. For example, in Mozart we have built a replicated transactional object store, called `GlobalStore` [128]. This keeps copies of a set of objects on several processes and gives access to them through a transactional protocol. The copies are kept coherent through the protocol. As long as at least one process is alive, the `GlobalStore` will survive.

Because of the failure detection provided by the `Fault` module, the Mozart system lets the `GlobalStore` and other fault-tolerant ADTs be written completely in Oz without recompiling the system. Ongoing research involves building abstractions for active fault tolerance and improved failure detection.

11.9 Security

An application is *secure* if it can continue to fulfill its specification despite intentional (i.e., malicious) failures of its components. Security is a global problem: a weakness in any part of the system can potentially be exploited. Security is a relative concept: no system is absolutely secure since with sufficient effort it can always be compromised. All we can do is increase the effort required to break the security, until it is not cost-effective for an adversary to attempt it. Security

issues appear at each layer of a distributed system. We identify the following layers [72]:

- **Application security.** This is a property of the application program. The application can continue to fulfill its specification despite adversaries whose attacks stay within the permitted operations in the application itself.
- **Language security.** This is a property of the language. In a secure language, applications can continue to fulfill their specifications despite adversaries whose attacks stay within the language. As we explain in Section 3.7.7, the kernel languages of this book provide language security because they have a rigorous semantics that permits the construction of secure ADTs.
- **Implementation security.** This is a property of the language implementation in the process. In a secure implementation, applications can continue to fulfill their specifications despite adversaries that attempt to interfere with compiled programs and the language's run-time system. Providing implementation security requires cryptographic techniques that are outside the scope of this book.
- **Operating system security, network security, and hardware security.** We group these three together, although each of them is a big topic that can be studied separately. The system is secure if applications can continue to fulfill their specifications despite adversaries who attempt to interfere with the operating system, the network, or the hardware. For the operating system and network, we can rely to some degree on off-the-shelf products. Hardware security is another matter entirely. Unless we have a special "hardened" computer, giving physical access to the computer always makes it possible to compromise security.

Each of these layers must be addressed to some degree, or otherwise the application is not secure. To judge how much effort must be put in making each layer secure, a *threat model* must be set up and a *threat analysis* done. Then a security policy must be defined, implemented, and verified. These activities are called *security engineering*. They are beyond the scope of this book. We recommend Anderson's book for an excellent overview [5].

Section 3.7 shows how to build secure abstract data types using language security. These techniques are necessary for building secure applications on the Internet, but they are not sufficient. We also have to address the other layers. For implementation security, we need a secure Mozart implementation. The development of such an implementation is ongoing research. Building implementation-secure systems is a research area with a long tradition. As an entry point in this area, we recommend the work on the E language and its secure implementation [123, 183].

11.10 Building applications

With the examples given in this chapter, you have enough technical knowledge already to build fairly sophisticated distributed applications.

11.10.1 Centralized first, distributed later

Developing an application is done in two phases:

- First, write the application without partitioning the computation between processes. Check the correctness and termination properties of the application on one process. Most of the debugging is done here.
- Second, place the threads on the right processes and give the objects a distributed semantics to satisfy the geographic constraints (placement of resources, dependencies between processes) and the performance constraints (network bandwidth and latency, machine memory and speed).

The large-scale structure of an application consists of a graph of threads and objects. Threads are created on the processes that need them. Objects may be stationary, mobile, or asynchronous. They exchange messages which may refer to objects or other entities. Records and procedures, both stateless entities, are the basic data structures of the application—they are passed automatically between processes when needed. Dataflow variables and locks are used to manage concurrency and dataflow execution.

11.10.2 Handling partial failure

The application must be able to handle partial failure. A good approach is to design for fault confinement. That is, to design the application so that failures can be *confined*, i.e., their effect will not propagate throughout the whole application but will be limited. Fault confinement has to be part of the initial application design. Otherwise the number of failure modes can be very large, which makes fault confinement infeasible.

There is a trade-off between the communication mode (synchronous or asynchronous) and the fault detection/confinement mechanism. Compared to synchronous communication, asynchronous communication improves performance but makes fault confinement harder. Consider a system with three active objects, T1, T2, and T3. T1 does an asynchronous send to T2 and continues, assuming that T2 is alive. Later, T1 sends a message to T3 under this assumption. But the assumption might have been wrong. T1 might have been executing for a long time under this wrong assumption. With synchronous sends this problem cannot occur. T1 does a synchronous send to T2 and is informed that T2 has a problem before continuing. This confines the fault to an earlier point of the program. The trade-off between early fault detection and asynchronous communication is fundamental, like the choice between optimistic and pessimistic concurrency control.

With asynchronous communication, the application must be prepared to correct any false assumptions it makes about the rest of the system working correctly.

There are different ways to realize fault confinement. One way is to build abstractions that do all the fault handling internally. If done well, this can hide completely the complexities of handling faults, at the cost of having to use the particular abstraction. The GlobalStore mentioned before takes this approach. If we cannot hide the faults completely, the next best thing is to have narrow interfaces (say, just one port) between processes. A final point is that a message-passing programming style is preferable over a shared-state style. Fault handling of distributed shared state is notoriously difficult.

11.10.3 Distributed components

Functors and resources are the key players in distributed component-based programming. A functor is stateless, so it can be transparently copied anywhere across the net and made persistent by pickling it on a file. A functor is linked on a process by evaluating it there with the process resources that it needs (“plugging it in” to the process). The result is a new resource, which can be used as is or linked with more functors. Functors can be used as a core technology driving an open community of developers who contribute to a global pool of useful components.

11.11 Exercises

1. **Implementing network awareness.** Explain exactly what happens in the network (what messages are sent and when) during the execution of the distributed lexical scoping example given in Section 11.4. Base your explanation on the distributed algorithms explained in Section 11.7.
2. **Distributed lift control system.** Make the lift control system of Chapter 5 into a distributed system. Put each component in a separate process. Extend the system to handle partial failure, i.e., when one of the components fails or has communication problems.
3. **A simple chat room.** Use the techniques of this chapter to write a simple server-based chat application. Clients connect to the server, receive all previous messages, and can send new messages. Extend your chat room to handle client failures and server failure. If there is a server failure, the client should detect this and allow the human user to connect to another server.
4. **A replicated server.** To make a server resistant to failures, one technique is to replicate it on two processes. Client requests are sent to both replicas, each of which does the computation and returns a result. The client needs

only to receive one result. This assumes that the server is deterministic. If one of the replicas fails, the other replica detects this, starts a new second replica using the `Remote` module, and informs the client. For this exercise, write an abstraction for a replicated server that hides all the fault handling activities from the clients.

Chapter 12

Constraint Programming

“Plans within plans within plans within plans.”
– Dune, *Frank Herbert* (1920–1986)

Constraint programming consists of a set of techniques for solving constraint satisfaction problems.¹ A *constraint satisfaction problem*, or *CSP*, consists of a set of constraints on a set of variables. A *constraint*, in this setting, is simply a logical relation, such as “X is less than Y” or “X is a multiple of 3”. The first problem is to find whether there exists a solution, without necessarily constructing it. The second problem is to find one or more solutions.

A CSP can always be solved with brute force search. All possible values of all variables are enumerated and each is checked to see whether it is a solution. Except in very small problems, the number of candidates is usually too large to enumerate them all. Constraint programming has developed “smart” ways to solve CSPs which greatly reduce the amount of search needed. This is sufficient to solve many practical problems. For many problems, though, search cannot be entirely eliminated. Solving CSPs is related to deep questions of intractability. Problems that are known to be intractable will always need some search. The hope of constraint programming is that, for the problems that interest us, the search component can be reduced to an acceptable level.

Constraint programming is qualitatively different from the other programming paradigms that we have seen, such as declarative, object-oriented, and concurrent programming. Compared to these paradigms, constraint programming is much closer to the ideal of declarative programming: to say *what* we want without saying *how* to achieve it.

Structure of the chapter

This chapter introduces a quite general approach for tackling CSPs called *propagate-and-search* or *propagate-and-distribute*. The chapter is structured as follows:

¹This chapter was co-authored with Raphaël Collet.

- Section 12.1 gives the basic ideas of the propagate-and-search approach by means of an example. This introduces the idea of encapsulating constraints inside a kind of container called a *computation space*.
- Section 12.2 shows how to specify and solve some example constraint problems using propagate-and-search.
- Section 12.3 introduces the constraint-based computation model and its two parts: constraints (both basic and propagators) and computation spaces.
- Section 12.4 defines computation spaces and shows how to program propagate-and-search with the computation space ADT.
- Section 12.5 shows how to implement the **choice**, **fail**, and **Solve** operations of the relational computation model with computation spaces.

12.1 Propagate and search

In this section, we introduce the basic ideas underlying the propagate-and-search approach by means of a simple example. Sections 12.3 and 12.4 continue this presentation by showing how the stateful computation model is extended to support this approach and how to program with the extended model.

12.1.1 Basic ideas

The propagate-and-search approach is based on three important ideas:

1. *Keep partial information.* During the calculation, we might have partial information about a solution (such as, “in any solution, X is greater than 100”). We keep as much of this information as possible.
2. *Use local deduction.* Each of the constraints uses the partial information to deduce more information. For example, combining the constraint “X is less than Y” and the partial information “X is greater than 100”, we can deduce that “Y is greater than 101” (assuming Y is an integer).
3. *Do controlled search.* When no more local deductions can be done, then we have to search. The idea is to search as little as possible. We will do just a small search step and then we will try to do local deduction again. A search step consists in splitting a CSP P into two new problems, $(P \wedge C)$ and $(P \wedge \neg C)$, where C is a new constraint. Since each new problem has an additional constraint, it can do new local deductions. To find the solutions of P , it is enough to take the union of the solutions to the two new problems. The choice of C is extremely important. A well-chosen C will lead to a solution in just a few search steps.

12.1.2 Calculating with partial information

The first part of constraint programming is calculating with partial information, namely keeping partial information and doing local deduction on it. We give an example to show how this works, using intervals of integers. Assume that x and y measure the sides of a rectangular field of agricultural land in integral meters. We only have approximations to x and y . Assume that $90 \leq x \leq 110$ and $48 \leq y \leq 53$. Now we would like to calculate with this partial information. For example, is the area of the field bigger than 4000 square meters? This is easy to do with constraint programming. We first declare what we know about x and y :

```
declare X Y in
X::90#110
Y::48#53
```

The notation `X::90#110` means $x \in \{90, 91, \dots, 110\}$. Now let us calculate with this information. With constraint programming, `xy > 4000` will return with true immediately:²

```
declare A in
A::0#10000
A=:X*Y
{Browse A>:4000}  % Displays 1
```

We can also display the area directly:

```
{Browse A}          % Displays A{4320#5830}
```

From this we know the area must be in the range from 4320 to 5830 square meters. The statement `A=:X*Y` does a constraint multiplication. Technically, it is called a *propagator*: it looks at its arguments a , x , and y , and propagates information between them. In this case, the propagation is simple: the minimal value of a is updated to 90×48 (which is 4320) and the maximal value of a is updated to 110×53 (which is 5830). Note that we have to give the initial information about a , for example that it is in the range from 0 to 10000. If we do not give this information, the constraint multiplication `A=:X*Y` will block.

Now let us add some more information about x and y and see what we can deduce from it. Assume we know that the difference $x - 2y$ is *exactly* 11 meters. We know this by fitting a rope to the y side. Passing the rope twice on the x side leaves 11 meters. What can we deduce from this fact? Add the constraint:

```
X-2*Y=:11
```

Technically, this new constraint is also a propagator. It does a local deduction with the information we know about x and y . The browser display is automatically updated to `A{5136#5341}`. This considerably increases the accuracy of

²The program fragment will display the integer 1, which means true. The boolean is given as an integer because we often need to do calculations with it.

our measurement: we know the area must be from 5136 to 5341 square meters. What do we know about x and y ? We can display them:

```
{Browse X}
{Browse Y}
```

This displays `X{107#109}` for x and `Y{48#49}` for y . This is a very simple example of calculating with partial information, but it can already be quite useful.

12.1.3 An example

We now look at an example of a complete constraint program, to see how propagate-and-search actually works. Consider the following problem:

How can I make a rectangle out of 24 unit squares so that the perimeter is exactly 20?

Say that x and y are the lengths of the rectangle's sides. This gives two equations:

$$\begin{aligned}x \cdot y &= 24 \\ 2 \cdot (x + y) &= 20\end{aligned}$$

We can also add a third equation:

$$x \leq y$$

Strictly speaking, the third equation is not necessary, but including it does no harm (since we can always flip a rectangle over) and it will make the problem's solution easier (technically, it reduces the size of the search space). These three equations are constraints. We call these equations *propagators*, since we will use them to make local deductions, i.e., “propagate” partial information about a solution.

To solve the problem, it is useful to start with some information about the variables. We bound the possible values of the variables. This is not absolutely necessary, but it is almost always possible and it often makes solving the problem easier. For our example, assume that X and Y each range from 1 and 9. This is reasonable since they are positive and less than 10. This gives two additional equations:

$$\begin{aligned}x &\in \{1, 2, \dots, 9\} \\ y &\in \{1, 2, \dots, 9\}\end{aligned}$$

We call these equations *basic constraints* since they are of the simple form “variable in an explicit set”, which can be represented directly in memory.

The initial problem

Now let us start solving the problem. We have three propagators and two basic constraints. This gives the following situation:

$$S_1 : \quad X*Y=:24 \quad X+Y=:10 \quad X=<:Y \quad || \quad X::1\#9 \quad Y::1\#9$$

which we will call the computation space S_1 . A *computation space* contains the propagators and the basic constraints on the problem variables. As in the previous example, we use the notation $X::1\#9$ to mean $x \in \{1, 2, \dots, 9\}$. We have the three propagators $X*Y=:24$, $X+Y=:10$, and $X=<:Y$. Syntactically, we show that these are propagators by adding the colon $:$ to their name.

Local deductions

Each propagator now tries to do local deductions. For example, the propagator $X*Y=:24$ notices that since Y is at most 9, that X cannot be 1 or 2. Therefore X is at least 3. It follows that Y is at most 8 (since $3*8=24$). The same reasoning can be done with X and Y reversed. The propagator therefore updates the computation space:

$$S_1 : \quad X*Y=:24 \quad X+Y=:10 \quad X=<:Y \quad || \quad X::3\#8 \quad Y::3\#8$$

Now the propagator $X+Y=:10$ enters the picture. It notices that since Y cannot be 2, therefore X cannot be 8. Similarly, Y cannot be 8 either. This gives:

$$S_1 : \quad X*Y=:24 \quad X+Y=:10 \quad X=<:Y \quad || \quad X::3\#7 \quad Y::3\#7$$

With this new information, the propagator $X*Y=:24$ can do more deduction. Since X is at most 7, therefore Y must be at least 4 (because $3*7$ is definitely less than 24). If Y is at least 4, then X must be at most 6. This gives:

$$S_1 : \quad X*Y=:24 \quad X+Y=:10 \quad X=<:Y \quad || \quad X::4\#6 \quad Y::4\#6$$

At this point, none of the propagators sees any opportunities for adding information. We say that the computation space has become *stable*. Local deduction cannot add any more information.

Using search

How do we continue? We have to make a guess. Let us guess $X=4$. To make sure that we do not lose any solutions, we need *two* computation spaces: one in which $X=4$ and another in which $X \neq 4$. This gives:

$$\begin{array}{ll} S_2 : & X*Y=:24 \quad X+Y=:10 \quad X=<:Y \quad || \quad X=4 \quad Y::4\#6 \\ S_3 : & X*Y=:24 \quad X+Y=:10 \quad X=<:Y \quad || \quad X::5\#6 \quad Y::4\#6 \end{array}$$

Each of these computation spaces now has the opportunity to do local deductions again. For S_2 , the local deductions give a value of Y :

$$S_2 : \quad X*Y=:24 \quad X+Y=:10 \quad X=<:Y \quad || \quad X=4 \quad Y=6$$

At this point, each of the three propagators notices that it is completely solved (it can never add any more information) and therefore removes itself from the computation space. We say that the propagators are *entailed*. This gives:

$$S_2 : \quad (empty) \quad || \quad X=4 \quad Y=6$$

The result is a *solved* computation space. It contains the solution $X=4 \quad Y=6$.

Let us see what happens with S_3 . Propagator $X*Y=:24$ deduces that $X=6 \quad Y=4$ is the only possibility consistent with itself (we leave the reasoning to the reader). Then propagator $X=<:Y$ sees that there is no possible solution consistent with itself. This causes the space to *fail*:

$$S_3 : \quad (failed)$$

A failed space has no solution. We conclude that the only solution is $X=4 \quad Y=6$.

12.1.4 Executing the example

Let us run this example in Mozart. We define the problem by writing a one-argument procedure whose argument is the solution. Running the procedure sets up the basic constraints, the propagators, and selects a distribution strategy. The distribution strategy defines the “guess” that splits the search in two. Here is the procedure definition:

```

proc {Rectangle ?Sol}
  sol(X Y)=Sol
in
  X::1#9    Y::1#9
  X*Y=:24   X+Y=:10   X=<:Y
  {FD.distribute naive Sol}
end

```

The solution is returned as the tuple `Sol`, which contains the two variables X and Y . Here $X::1\#9$ and $Y::1\#9$ are the two basic constraints and $X*Y=:24$, $X+Y=:10$, and $X=<:Y$ are the three propagators. The `FD.distribute` call selects the distribution strategy. The chosen strategy (*naive*) selects the first non-determined variable in `Sol`, and picks the leftmost element in the domain as a guess. To find the solutions, we pass the procedure to a general search engine:

```
{Browse {SolveAll Rectangle}}
```

This displays a list of all solutions, namely `[sol(4 6)]` since there is only one.

All the constraint operations used in this example, namely `::`, `=:`, `<:`, and `FD.distribute` are predefined in the Mozart system. The full constraint programming support of Mozart consists of several dozen operations. All of these

operations are defined in the constraint-based computation model. This model introduces just two new concepts to the stateful concurrent model: finite domain constraints (basic constraints like $x : 1\#9$) and computation spaces. All the richness of constraint programming in Mozart is provided by this model.

12.1.5 Summary

The fundamental concept used to implement propagate-and-search is the *computation space*, which contains *propagators* and *basic constraints*. Solving a problem alternates two phases. A space first does local deductions with the propagators. When no more local deductions are possible, i.e., the space is *stable*, then a search step is done. In this step, two copies of the space are first made. A basic constraint C is then “guessed” according to a heuristic called the *distribution strategy*. The constraint C is then added to the first copy and $\neg C$ is added to the second copy. We then continue with each copy. The process is continued until all spaces are either *solved* or *failed*. This gives us all solutions to the problem.

12.2 Programming techniques

Now that we have seen the basic concepts, let us see how to program with them. A constraint problem is defined by a one-argument procedure. The procedure argument is bound to the solution of the problem. Inside the procedure, next to the usual language operations, two new kinds of operations are possible:

- Constraints. These specify the relationships between the different parts of the problem. They can be either basic constraints or propagators.
- Specification of the distribution strategy. This specifies how the search tree is to be formed, i.e., which constraints C and $\neg C$ are chosen at each node when doing a search step.

In contrast to relational programming (see Chapter 9), there is no explicit creation of choice points (no **choice** statement). This would be too crude a way to search; what actually happens is that choice points are created dynamically in terms of the distribution strategy that is specified.

12.2.1 A cryptarithmic problem

Now that we have the basic concepts, let us see how we can program with them. As example we take a well-known combinatoric puzzle, the *Send+More=Money* problem.³ The problem is to assign digits to letters such that the following addition makes sense:

³This example is taken from [174].


```

proc {SendMoreMoney ?Sol}
  S E N D M O R Y
in
  Sol=sol(s:S e:E n:N d:D m:M o:O r:R y:Y) %1
  Sol:::0#9 %2
  {FD.distinct Sol} %3
  S\=:0 %4
  M\=:0
  1000*S + 100*E + 10*N + D %5
  + 1000*M + 100*O + 10*R + E
  =: 10000*M + 1000*O + 100*N + 10*E + Y
  {FD.distribute ff Sol} %6
end

```

Figure 12.1: Constraint definition of *Send-More-Money* puzzle

$$\begin{array}{rcccccc}
 & S & E & N & D & & \\
 + & M & O & R & E & & \\
 \hline
 M & O & N & E & Y & &
 \end{array}$$

There are two conditions: each letter is assigned to a *different* digit and the leading digits of the numbers are different from zero ($S \neq 0$ and $M \neq 0$).

To solve this problem with constraints, the first step is to *model* the problem, i.e., to set up data structures and constraints that reflect the problem structure. In this problem, it is easy: each digit is a variable and the problem conditions become constraints on the variables. There are eight different letters, and therefore eight variables.

The second step is to define a one-argument procedure that implements this model. Figure 12.1 shows one way to define the procedure. The numbered statements have the following effects:

1. The solution `Sol` is a record with one field for every different letter.
2. The fields of `Sol` are integers in the domain $\{0, \dots, 9\}$.
3. The fields of `Sol` are pairwise distinct, i.e., no two have the same value.
4. Since they are leading digits, the values of `S` and `M` are not zero.
5. All the digits satisfy the equation $SEND + MORE = MONEY$.
6. The distribution strategy tries the letters according to a *first-fail* strategy (`ff`). This means that the strategy tries first the letter with the least number of possibilities, and with this letter it tries the least value first.

The third step is to solve the problem:

```
{Browse {SolveAll SendMoreMoney}}
```

This computes and displays a list of all solutions. Note that this is done in the same way as search in relational programming (see Chapter 9). This displays:

```
[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

In other words, there is just one solution, which is:

$$\begin{array}{r} 9 \ 5 \ 6 \ 7 \\ + \ 1 \ 0 \ 8 \ 5 \\ \hline 1 \ 0 \ 6 \ 5 \ 2 \end{array}$$

That is all there is to it! In practice, things are a bit more complicated:

- **Modeling the problem.** Modeling the problem is not always easy. Often there are many possible ways to represent the problem in terms of constraints. It is not always obvious which one is best!
- **Constraints and distribution strategies.** There are many constraints and distribution strategies to choose from. Which ones are best depends strongly on the problem.
- **Understanding the problem.** The first solution to a realistic problem is usually too inefficient. There are many techniques to improve it. Some possibilities are to take advantage of problem structure, to use redundant constraints, to use different distribution strategies, and to use the Explorer (an interactive graphical search tree exploration tool, see [171]).

12.2.2 Palindrome products revisited

In Section 9.2.1, we saw how to find palindrome products with relational programming. The technique used there takes 45 seconds to find all solutions for 6 digit palindromes. Here is a smarter solution that takes advantage of constraints and the propagate-and-search approach:

```
proc {Palindrome ?Sol}
  sol(A)=Sol
  B C X Y Z
in
  A::0#999999 B::0#999 C::0#999
  A=:B*C
  X::0#9 Y::0#9 Z::0#9
  A=:X*100000+Y*10000+Z*1000+Z*100+Y*10+X
  {FD.distribute ff [X Y Z]}
end
```

This takes slightly less than two seconds. We can do even better by realizing that a palindrome $XYZZYX$ is always a multiple of 11. That is, $XYZZYX = X \cdot 100001 + Y \cdot 10010 + Z \cdot 1100$, which means $XYZZYX/11 = X \cdot 9091 + Y \cdot 910 + Z \cdot 100$. Taking advantage of this, we can specify the problem as follows:

```

proc {Palindrome ?Sol}
    sol(A)=Sol
    B C X Y Z
in
    A::0#90909 B::0#90 C::0#999
    A=:B*C
    X::0#9 Y::0#9 Z::0#9
    A=:X*9091+Y*910+Z*100
    {FD.distribute ff [X Y Z]}
end

```

This takes slightly less than 0.4 seconds to solve the same problem. What can we conclude from this simple example? Many things:

- A constraint-based formulation of a combinatoric problem can be much faster than a generate-and-test formulation. For palindrome product, the constraint solution is more than 100 times faster than the naive solution.
- To make it fast, you also have to take advantage of the problem structure. A little bit of smarts goes a long way. For palindrome product, taking advantage of the solution being a multiple of 11 makes the program 5 times faster.
- A fast solution is not necessarily more complicated than a slow solution. Compare the slow and fast solutions to palindrome product: they are about equal in length and ease of understanding.
- Performance can depend strongly on the *exact* problem formulation. Changing it a little bit can make it much faster or (usually) much slower.
- To write a good specification, you have to understand the operational meaning of the constraints as well as the logical meaning. The latter is enough for showing correctness, but the former is essential to get good performance.

12.3 The constraint-based computation model

The propagate-and-search approach is supported by adding two concepts to the stateful concurrent model: basic constraints and computation spaces. Basic constraints are a simple generalization of declarative variables in the single-assignment store. Computation spaces extend the model as shown in Figure 12.2.

A computation space collects together basic constraints and propagators, as we saw in the example of Section 12.1.3. The basic constraints are a constraint store. The propagators are threads. A computation space is always created inside a parent space; it can see the constraints of its parent. In the figure, *x* is bound to a computation space that is created inside the top-level space.

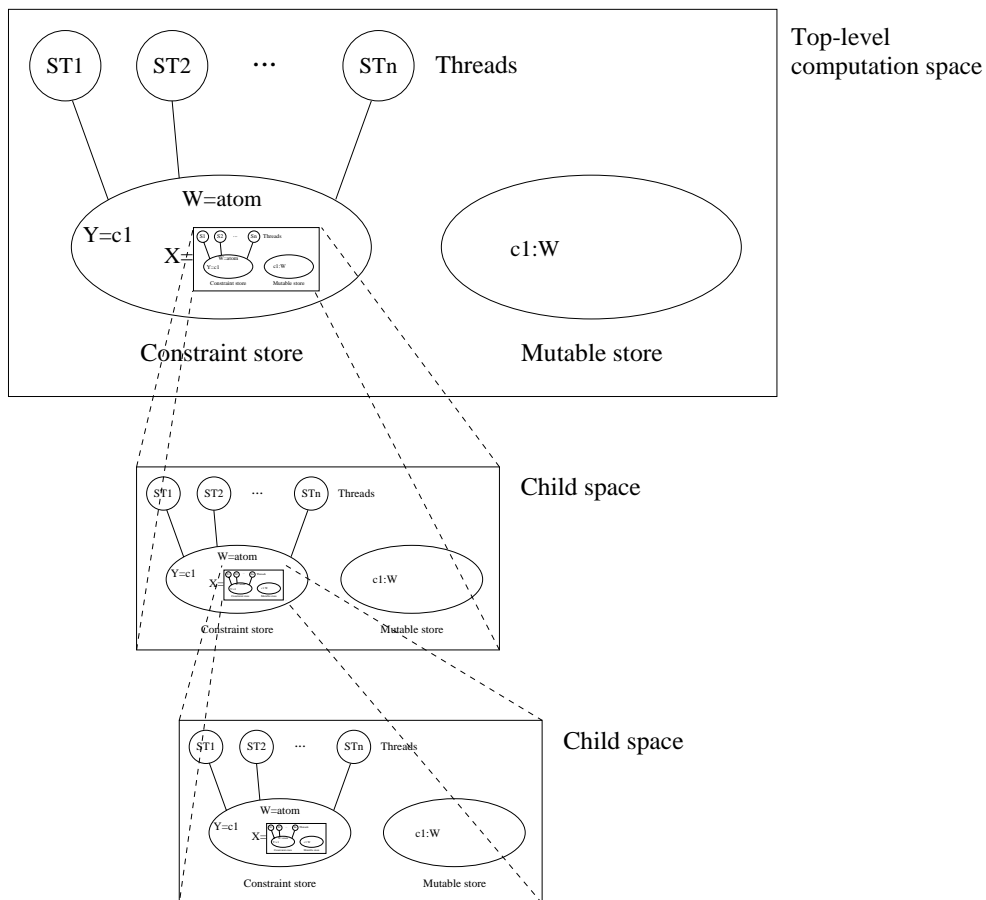


Figure 12.2: Constraint-based computation model

A computation space is also an ADT that implements a number of operations. With these operations, we can implement the propagate-and-search technique of Section 12.1. The operations are explained in Section 12.4.

12.3.1 Basic constraints and propagators

We introduce basic constraints for *finite domains* as equations of the form $x \in D$, where D is a finite integer set. This partial information is told to the store by the statement $x :: D$. The domain D is specified with a compact notation (see the examples in the previous sections). Successive tells $x :: D_1, x :: D_2, \dots, x :: D_n$ restricts the domain of x to $D_1 \cap D_2 \cap \dots \cap D_n$, provided that the latter is nonempty. Telling the empty domain for a variable would result in an inconsistent store (do you see why?), so such a tell must fail. The basic constraint $x \in \{n\}$ is simplified to the equivalent relation $x = n$.

The usual variable declaration does not tell explicitly the domain of a fresh variable. Its domain is implicit: it is the domain of rational trees. A rational tree is a non-partial value build with records and other basic values. Equality with partial values acts as a domain restriction. For instance, telling $x = \text{person}(\text{name}:y \text{ age}:z)$ restricts the domain of x to the rational trees that match the partial value $\text{person}(\text{name}:y \text{ age}:z)$.

A propagator is simply a thread that tells domain constraints to the store according to their semantics. Each time a variable's domain is changed in the store, the propagators that use that variable must be given a chance to execute, so they can propagate new partial information to variables. Waiting for a domain change is a fine-grained variant of waiting for determinacy. A multiset of propagators must behave in a concurrent declarative fashion, because that makes *controlled search* effective.

12.4 Computation spaces

In the previous sections we have seen how to use constraints with built-in distribution strategies. In this section we explain how computation spaces work, and how to program search engines and distribution strategies with them.

Computation spaces are an abstraction that permits the high-level programming of search abstractions and deep guard combinators. With computation spaces, the computation model looks something like Figure 12.2. All the search abstractions of Chapters 9 and 12 are programmed using spaces. Spaces have the flexibility needed for real-world constraint problems and they can be implemented efficiently: on real-world problems the Mozart implementation using copying and recomputation is competitive in time and memory use with traditional systems using trailing-based backtracking [168].

This section defines computation spaces, the operations that can be performed on them (see Table 12.1), and gives an example of how to use them to program

search. Actually we use the example as a roadmap throughout the definitions of concepts and operations. The discussion in this section follows the model in [172, 169]. This model is implemented in the Mozart system [129] and refines the one presented in the articles [167, 170]. The space abstraction can be made language-independent; [78] describes a C++ implementation of a similar abstraction that provides both trailing and copying.

12.4.1 Programming search with computation spaces

A *search strategy* defines how the search tree is explored, e.g., depth-first search or breadth-first search. A *distribution strategy* defines the shape and content of the search tree, i.e., how many alternatives exist at a node and what constraint is added for each alternative. Computation spaces can be used to program search strategies and distribution strategies independent of each other. That is, any search strategy can be used together with any distribution strategy. Here is how it is done:

- Create the space with the correct program inside. This program defines all the variables and constraints in the space.
- Let the program run inside the space. Variables and propagators are created. All propagators execute until no more information can be added to the store in this manner. The space eventually reaches stability.
- During the space's execution, the computation inside the space can decide to create a choice point. The decision which constraint to add for each alternative defines the distribution strategy. One of the space's threads will suspend when the choice point is created.
- When the space has become stable, execution continues outside the space, to decide what to do next. There are different possibilities depending on whether or not a choice point has been created in the space. If there is none, then execution can stop and return with a solution. If there is one, then the search strategy decides which alternative to choose and commits to that alternative.

The next section explains the operations we need for this approach, together with a concrete example of a search engine. Section 12.5 gives another example of how to program search with spaces. Many other strategies can be programmed than are shown here; for more information see [172, 169].

12.4.2 Definition

Our goal is to present computation spaces as a mean for implementing search strategies and distribution strategies. We will explain in detail the execution of a concrete example of a search engine on a small problem. The definitions of concepts and operations will be given as they come in the execution.