6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Distributed Computing Based on Clean Dynamics*

Hajnalka Hegedűs, Zoltán Horváth

Department of Programming Languages and Compilers, Faculty of Informatics,
Eötvös Loránd University, Budapest
e-mail: heha@inf.elte.hu, hz@inf.elte.hu

Abstract

Clean is a lazy, pure functional language. The relatively new language construct, called `dynamic`, allows of sending, storing and reloading data and even functions in a type safe way. Thus Clean can be used as the base of writing distributed, mobile code. Our aim is to develop a middleware based on Clean dynamics which supports writing distributed functional programs such as for computations in Grid environment. Our goal is to support location, migration, persistence and failure transparency. An important issue is scalability. The proposed system supports asynchronous and synchronous communication and property-carrying code.

This paper is a requirement and feasibility analysis and compares the system with the corresponding language elements of JoCaml (a language for concurrent distributed mobile programming based on non-pure functional language Caml).

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems; D.1.1 [Programming Techniques]: Applicative (Functional) Programming;

Key Words and Phrases: mobile code

1. Introduction

This paper is a requirement and feasibility analysis and compares the system with the corresponding language elements of JoCaml (a language for concurrent distributed mobile programming based on non-pure functional language Caml).

In distributed functional programming one uses a functional computational language and a distinct control language. In our case computational language is Clean. We will design a Clean-based concurrent control language.

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T037742 and the Bolyai Research Scholarship.

Agents provide a flexible way to use data stored on remote servers or write mobile code. They can move around network nodes and collect data, perform operations on each node or simply supply a service. This indicates they are really useful when it comes to distributed computation so we will use agents in our system. The base of our agents is a Clean `dynamic` which contains the (type safe) data and code of the agent. Our agents also have an execution state which stores the values of executed subexpressions, parameters etc. and a control state which contains general data on the agent (see figure 4). In our system one can also include semantical properties in the agent which can be checked by the recipient who than can decide to let the agent execute or not.

For the understanding the further sections of the paper, we have to spear a few words on JoCaml agents as well. JoCaml have language structures called `location`-s, which can be organised to a hierarchical structure. Each location can have sublocations and a location can migrate under an other location whenever its necessary.

Returning to our system, our language extension should contain primitives for building and migrating and receiving an agent. It is also important to let the agent initiate its migration. Though language primitives are not sufficient to write distributed applications. We also need a middleware to support data sharing and communication. The middleware should take care of restarting agents when the underlying system crashes, keep track of the shared resources etc. It also has to hide the heterogeneous architecture and provide a uniform way of communication.

In this paper we draw up the requirements against on our system, sketch a way how can these requirements can be met and describe how JoCaml meets or doesn't meet them.

2. Requirements for Agent Languages

There are three requirements, an agent language should meet [3]. These are the following:

- code manipulation tools,
- heterogeneity,
- performance.

Code manipulation tools provide means for identifying, transferring and executing agents. Identifying the code of an agent is essential when we need to migrate the agent and not as trivial as it seems. The agent should be self containing enough to be able to run at the remote site where it migrates. In JoCaml, a location can call back to its originating site, and is able to use every function which was local to it at the place of its definition. See 2.1 as an example [4] of an agent calling back to its originating site.

Example 2.1.

```
# let def new_cell there =
#    let def log s = print_string ("cell "^s^"\n"); reply in
#    let loc applet
#       def get() | some! x = log ("is empty"); none(); reply x
#       and put x | none!() = log ("contains "^x); some x | reply
#       do {go there; none ()} in
#    reply get,put
# ;;
```

The example defines a buffer, `new_cell` which can contain one value and has a destructive read. This means, that in parallel with a reading operation the buffer becomes empty. The operations of the buffer are defined within a location, `applet`. This location moves under an other location (`there`) which is given as a parameter for `new_cell`. The definition of `get` and `put` is irrelevant from our point of view, the only thing the reader should consider, that both definitions have calls to `log`, so we can track the usage of the cell from the originating location. `log` will be accessible even when it migrates under location `where`. So we can conclude that when a location migrates it keeps its original communication capabilities unaffected. Thus the code of an agent will only contain the code defined within a particular location. When transferring a location, we don't need to pack anything else together with the code, the runtime system will take care of the existing communication links so everything the agent was able to use at its originating site, will be accessible from the new site as well. After a location migrates under an other location, it behaves like a local function so there is no need to have distinct primitives to execute it.

Callbacks to the originating site has drawbacks. One is that execution time can increase painfully when a location has many callbacks to its originating site and the underlying network is slow. The other drawback is that the originating site has to be online all the time. If network connection breaks, the agent will fail. Thus in our system, we try to avoid callbacks as much as possible. The base of an agent in our system is a `dynamic`, which can be packed and sent to an other site using the `sendDynamic` and `receiveDynamic` functions of our experimental Clean version [6]. To avoid callbacks, everything which is referenced from the dynamic, must be packed together with it. This means that `sendDynamic` and `receiveDynamic` needs to compute the transitive closure of reference chains. After receiving a dynamic, one can use it as a local data.

Heterogenity is also an important issue in designing an agent language. JoCaml supports Unix and WindowsNT platforms. Since the WindowsNT version is based on Cygwin, it is like enough that agents can migrate from one platform to other with no difficulty.

Clean supports a bit more platforms: Windows, Linux, Solaris and PowerMac. Since the representation of `dynamic` contains symbolic native code, its very likely to be transferable between platforms based on the same architecture (e.g. Windows and Linux ix86). Although it might be very difficult to transfer code to the other architectures so the prototype of our system will only support Windows and Linux.

Its really hard to measure performance. Good performance means that the code of agents should be kept reasonably small (which is easy in JoCaml because of the presence of callbacks but a bit more complicated in Clean). We also need agents to be easily transferable and executable. It always depends on the application which aspect is the most important. For example when we have an overladen site which delegates work to other sites once in a while, its better to have large but self containing agents then using callbacks and increasing the load of the already overladen site.

Clean's self containing agents can be really large since normally we would need to pack the whole standard environment together with the `dynamic`. Though the sender and the receiver can communicate, and if the receiver has the correct version of a function used by the `dynamic`[1], we wouldn't need to pack the function together with the code, the recipient's copy can be used instead.

3. Optional Elements of Agent Languages

There are features that are not essential for agent languages but can make life easier. These are the following [3]:

- remote resource access,
- strong typing,
- automatic memory management,
- stand alone execution,
- security.

Remote resource access in this respect means, that the agent should be capable of using the resources of their eventual execution environment. In JoCaml, one can use any registered resource after lookup (see example 3.1) and any resource which was local to its definition (as in example 2.1).

Example 3.1. [4]

```
# spawn{ let def f x = reply x*x
#        in Ns.register "square" f vartype; };;
#
# spawn{ let sqr = Ns.lookup "square" vartype
#         in print_int (sqr 2);;
```

In this example one process registers function `f` on a name `square`. The other process looks it up from the name server and uses it in printing the square of 2. This also works when the processes run on different machines.

[1] This can be easily decided by using MD5 checksums in the name of dynamic files [6].

In Clean resources (such as the display or files) can be accessed using unique types. Currently dynamic and unique types can't be used together (e.g. a dynamic can't contain a unique type), because of the differences between the two file systems. There is a research in progress to make this feature available [7].

Strong typing provides a way of low level semantical checking. It helps the programmer to find errors which are otherwise really hard to track, and increases the reliability of the program. Both systems provides strong typing. Using Clean dynamics one can even send data in a type-safe way.

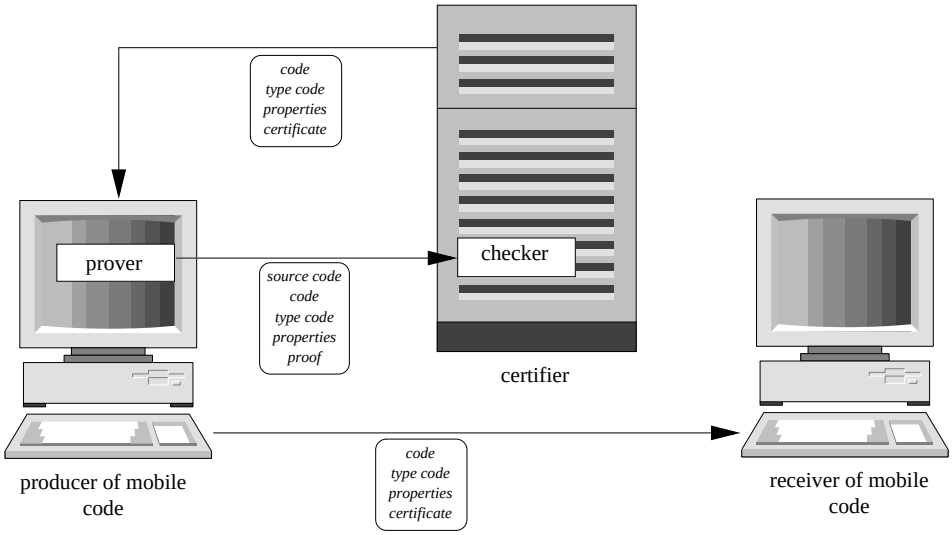
Automatic memory management is also important for the performance of an agent system. It reduces the possibility of memory leaks and also helps to avoid several programming errors (because of the lack of explicit memory allocations). This is also present in both languages, and our experimental Clean version also has a dynamic garbage collector [6], which collects and deletes the code of `dynamic`-s which are not referenced by any other `dynamic`. The garbage collection runs locally on each node participating in the system. This allows us to save space on a reasonably small price (no network communication is needed for garbage collection).

Stand alone execution means that agents don't have implicit callback to their originating site. When a callback is necessary for the correct operation of agent, the programmer has to state it explicitly in the code. This means that the agent should be able to run even when there is no connection with its originating site. As an example let us consider an agent which is delegated from a laptop computer. The agents task is to update a distributed database using the data contained in it. In this case the agent travels around the nodes of the database and does the necessary modifications. The originating computer can go offline since there is no need to communicate with it during the update. Later the computer can join the network again, and the agent can send it a report on the status of the update.

In JoCaml the programmer explicitly defines callbacks by defining functions outside the location but within the same scope (see example 2.1) or by using lookup. In our experimental Clean, every function needed is sent together with the agent, so it doesn't have to call back to its original location.

The last but probably the most important aspect is security. In our point of view security means protection from malicious or erroneous code. In our experimental system we use Certified Proved Property Carrying Code (CPPCC) [5] for guaranteeing good-behaving code. Figure 3 presents a sketch of the system.

At first the programmer should state semantical properties of the dynamic code. These has to be verified (using the Clean-based verification system, Sparkle). After that the source code, the native code, the type code (recall that the type code bears low lever semantical information), the properties, and the sketch of the proof are packed together and sent to a certifier. The certifier checks if the native code is generated from the presented source code and if the properties hold for the given code (using the proof sketch). If it finds everything correct, it generates a certificate, and packs it together with the native code, the type code and the properties. This packet is sent back to the writer of the dynamic, who can then



send it to whoever he wants. The receiver of the code can check the certificate, the type information and the attached properties to decide if the code is correct and fits his requirements. This method allows us to use not even type safe code, but code which is guaranteed to do what we expect it to do, which provides a way of writing highly reliable systems. Unfortunately JoCaml doesn't have any security features at all.

4. Features Needed for Distributed Computation

For writing a distributed application, language primitives supporting agents are usually not enough. We need features supporting distributed computation such as particular kinds of transparency and scalability. We will use the definitions of [2]. Providing these features is usually the task of the middleware. We have chosen to implement the following features:

- location transparency,
- migration transparency,
- failure transparency,
- scalability in size,
- administrational scalability.

Location transparency means that the user of a resource or agent doesn't need to know anything about the location of the resource or agent. This transparency can be achieved by using a name server. That's exactly as it is in JoCaml (see

example 3.1). When a resource needs to be shared, it can be registered to a name server. After that everyone connected to that name server can lookup the resource. We would like to use the same approach in our system as well.

Migration transparency means that when a resource or an agent moves, the users of it doesn't need to know about the migration. They should be able to use the agent/resource as before. To achieve this goal, the name server should keep track of the registered agents/resources. Normally an agent/resource can report its location after a successful migration. It is also advisable to report before the migration, because if the migration is unsuccessful, the name server initiate the resend of the agent. In this case we need to have the copy of the agent at the original place until the migration completed, though it's recommended to suspend the execution of it while the migration is in progress.

Failure transparency means that the user of an agent doesn't need to know about the failure of the agent in case it is not the fault of the agent, for example when the machine where the agent runs crashes. In JoCaml one have only limited possibilities of exception handling: one can halt a location and all of its sublocations using the `halt` primitive. This is a very radical way to handle exceptions and it lacks the potential of recovery. A bit more sophisticated way to handle exceptions is to use the `fail` primitive. `fail` allows the programmer to detect the failure of a location and run a process for recovery. This is not sufficient for failure transparency though can increase the reliability of programs.

What we need is automatic recovery. There are several situations where an agent can fail:

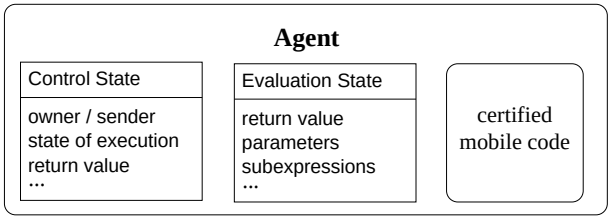
The site crashes before the agent started to execute. In this case we simply have to start the execution of the agent when the site recovers.

The site crashes while the agent executing. In this case we have to restart the agent. We can also use the (possibly) stored subresults of the previous execution.

The execution of the agent is complete, but further actions needed. In this case if the agent has to report back somewhere or migrate we can send the report message or the agent to the appropriate site when the site recovers.

The execution of the agent is complete and no further actions needed. In this case the agent can simply be dropped.

To handle these situations we need special states for our agents (see figure 4). Our agent have a control state. It consist information on the state of execution of the agent, the further actions needed and the owner of the sender. The state of execution tells the middleware that the agent started or completed its execution and if the further actions (eg. reporting back or migrating) are completed or not. This information can be used to decide which of the described situations occurred when the site crashed. If the crash occurred when the execution was in progress, the middleware can use the data stored in evaluation state to give the agent the already computed values.



Scalability have two aspects in our case. When the number of nodes joining the distributed system grows, the can cause a bottleneck if only a single name server present (the same holds for the dynamic code library). To avoid bottleneck we need to have a distributed name server. In this case we need to keep the data stored on the different servers consistent.

The other aspect is administrational scalability. In our case it is likely that a new organisation would like to join the distributed system. We need to protect the nodes of different organisations from each other. For example one can have data which is private for his organisation but shared with the nodes of the same organisation. We don't want this data to be visible for the nodes of the other organisation. To provide this, we need to attach security information to any shared resources (eg. who can access the data and what kind of actions can he perform). This data can be stored in XML.

5. Conclusions

In our paper we overviewed the distributed and agent language aspects of JoCaml and the potentials of Clean which can make it a base of a distributed agent based system. As we seen JoCaml misses some features we need. JoCaml doesn't have any features supporting security and has only very limited tools to achieve failure transparency. We discussed how these (and several more) features can be implemented in Clean and gave an outline of our system. We introduced a sketch of agents which are suitable for our goals.

Though there are features Clean already has (eg. code manipulation tools, strong typing, automatic memory management and stand-alone execution), there is still a lot of work to be done. We have to improve features which are partly implemented in prototypes or not sufficient yet such as security tools, and we have to implement some features from scratch, like location, migration and failure transparency and remote resource access. After having all the necessary language primitives and the middleware, we will have a powerful highly reliable system which can support the work of programmers a great deal.

References

[1] Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0. Language Report*, Draft, University of Nijmegen, 2001.

[2] Tanenbaum, A.S., van Steen, M..: *Distributed Systems. Principles and Paradigmes.*, Prentice-Hall, 2001.

[3] Knabe, F. C.: *Language Support for Mobile Agents.* PhD. thesis, Carnegie Mellon University, Pittsburgh, 1995.

[4] Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: *JoCaml: a Language for Concurrent Distributed and Mobile Programming*, Preliminary Proceedings for the Fourth Summer School on Advanced Functional Programming, St Anne's College, Oxford, 2002, pp. 1-25.

[5] Daxkobler, K., Horváth, Z., Kozsik, T.: *A Prototype of CPPCC - Safe Functional Mobile Code in Clean.* Proceedings of Implementation of Functional Languages'02, Madrid, Spain, Sept. 15-19, 2002. pp. 301-310.

[6] Ivicsics, M.: *The Dynamic Type System of Clean.* Scientific Students' Associations Conference, ELTE, Budapest, December, 2002.

[7] Dutot, P.-F.: *Interaction between Dynamics and the Uniqueness typing system* Report of an internship in the University of Nijmegen, 1999.