

# CORRELATE: High-Level Support for Traveling Agents

Wouter Joosen\*, Frank Matthijs, Johan Van Oeyen, Bert Robben†,  
Stijn Bijmens and Pierre Verbaeten

Dept. of Computer Science, K.U.Leuven  
Celestijnenlaan 200A  
B-3001 Leuven  
Belgium  
email: Wouter.Joosen@cs.kuleuven.ac.be

CW Report 236, Version 2.1

October 21, 1996

## *Abstract:*

CORRELATE is a concurrent object-oriented language with a flexible run time system that enables the instantiation of application specific run time objects. Recently we have exploited this capability in the development of mobile agents for large scale distributed computing systems, such as the Internet.

In this report, we describe the main ingredients of CORRELATE and its run time system. We illustrate how this architecture supports mobile agents. We discuss the ongoing work in improving the applicability of the environment.

## **1 Introduction**

The concept of an agent is evolving to become widely applied in the development of new application software, ranging from entities such as mobile Internet agents that search (and possibly buy) information on the web, over user interface agents that attempt to deliver a personalized view on information repositories, to distributed AI software in which large collections of agent-based entities cooperate to expose emergent behaviour.

---

\*Researcher for the Flemish I.W.T.

†Research Assistant of the Belgian National Fund for Scientific Research

Our main interest in this report is to discuss agents in the context of large-scale heterogeneous computer networks (such as campus-wide information systems, telecom backbones or the Internet). The goal of the CORRELATE project however, is to build a language that acts as a high level programming interface and that offers an open execution platform for a wide variety of agent-based applications. Our basic view on an agent therefore is broad and relatively straightforward: agents are active entities (objects) with some degree of autonomy. Unless explicitly stated differently, objects are supposed to be active.

## 1.1 Developing Applications with Mobile Agents

The computer architecture on which an agent runs ranges from a PC to the whole Internet. In the latter case, the application developer not only deals with the logical distribution of information (amongst agents), but also with distributed architecture that shows through when it comes to obtain a reliable and well-performing application. Enhancing agents with the capability to move in the distributed architecture is a key ingredient to obtain satisfying performance. We first sketch our view on the development process for mobile agents.

The development of mobile agent application software is inherently difficult, and a clean and powerful program development process is essential to achieve reliable and efficient software that remains portable, maintainable and extensible. In our opinion, the development process should be tackled in two phases. First, the application developers describe a high-level model of the application: they describe the application a domain as a set of *autonomous and active* objects. Only a *computational* view of the application is presented: interacting agents live in a global object space. This concept effectively hides distribution through location independent object invocation. Secondly, application developers describe a *physical* view of the global object space: an optimal execution environment is instantiated. This execution environment implements the global object space and targets the application to the distributed architecture. In this phase the application programmer can instantiate a location control subsystem to improve performance, or a fault tolerance subsystem to increase reliability. Note that a subsystem may only affect a single application object/agent.

We have developed an open framework to guide the application programmer in the above mentioned steps. Various support systems can be instantiated to take application specific requirements into account. The focus of this report is on location control subsystems.

## 1.2 Traveling Agents and a Separation of Concerns

The view that is represented above may surprise someone who wants to develop a mobile agent because location transparency seems a property that is non-existent in this case. However, as argued in this report, we believe that it is appropriate to maintain a maximal amount of modularity to guarantee reusability.

In our model, a mobile agent consists of two objects: a pure application object (describing what to do) and a location control subsystem (determining where to do it). We

```

active Shopper{
...
interface: // Reactive Behaviour
    void ShowTicket();
behaviour:
    bool IsTicketBought();
    for ShowTicket() precondition IsTicketBought();
implementation:
    ...
};

```

Figure 1: A Shopper

have built a prototype that is based on a meta-level architecture with an explicit metaobject protocol. In our opinion, this approach has proven to be a strong basis for integrating mobility support (both mechanisms and policy) with the computational view on an agent.

As a result, different location control policies are offered in libraries with reusable components. An application programmer will be enabled to choose appropriate subsystems instead of creating them from scratch.

The rest of the report is structured as follows. The next section introduces CORRELATE agents, which are concurrent objects that have autonomy. Section 3 outlines how agent applications are developed by exploiting the architecture of the language run-time. Section 4 discusses how we have tackled security problems. Section 5 covers the status of our project and contrasts CORRELATE with some other languages for mobile agents (typically for the Internet). Section 6 is a summary.

## 2 A CORRELATE Intro

In this section, we will briefly sketch how CORRELATE supports concurrent object-oriented technology. CORRELATE is a class-based concurrent object-oriented language. The CORRELATE syntax is influenced by C++: our prototype is built as a preprocessor for C++. We will only discuss language annotations as far as they are specifically related to concurrency and distribution.

A CORRELATE class mainly is a code template that enables the instantiation of individual agents<sup>1</sup>. In the computational view, the focus is on location independent interaction in a global agent space.

In its basic form, a CORRELATE agent is pretty much comparable with a concurrent object. An important concern in concurrent environments is the synchronisation of objects. Depending on the state of the object, a specific operation may or may not be

---

<sup>1</sup>The code fragments that will be shown through this paper are based on CORRELATE classes, because they essentially model large collections of similar agents.

```

active Shopper{
  autonomous: // Autonomous behaviour
    void SearchTicket(RealWorldPlace departure,RealWorldPlace destination);
  interface: // Reactive Behaviour
    void ShowTicket();
  behaviour:
    bool IsTicketBought();
    for ShowTicket() precondition IsTicketBought();
  implementation:
    ...
};

```

Figure 2: An Autonomous Shopper

executed. In other words, in a certain state an object can accept only a subset of its entire set of operations in order to maintain its internal integrity. This issue can be expressed with synchronisation constraints that reflect application specific semantics of the object. The description of synchronisation constraints inherently enforces the programmer to reveal state information of an active object. The problem of specifying synchronisation constraints therefore is related to the inherent conflict between encapsulation (one of the basic features of OO) and concurrency.

Our basic approach in designing the CORRELATE language is to define a view on an active object that reveals more state information than the amount that would be required in a sequential model, while maintaining encapsulation as much as possible. CORRELATE objects therefore expose an intermediate abstraction layer which basically corresponds to an abstract state machine.

At the level of a class definition, the language's syntax provides a behaviour section that describes the abstract states. Figure 1 describes a Shopper agent that buys airplane tickets for its owner.

In principle, each abstract state is represented by a boolean selection operation that can determine whether an object is in the corresponding state or not. For the shopper, *IsTicketBought* determines whether the agent has actually bought the ticket. The application programmer then implements the mapping of the actual implementation (the private data members) on the abstract states. A precondition can be specified for each constrained operation. This precondition can use the abstract state and the parameters of the operation. When a precondition is just a boolean expression that uses abstract states, it can be inserted in the behaviour section of the class header. (This has been illustrated in Figure 1 for the *ShowTicket* operation: the agent can only show the ticket when it has actually bought it.) Otherwise it is coded as any other operation of the class. Operations without a precondition are unconstrained.

A main feature of a CORRELATE agent is its capability to perform so-called autonomous operations. Autonomous operations reflect the autonomy of the instantiated

```

active TravelAgency{
  autonomous: // Autonomous behaviour
    void AnnouncePrices();
  interface: // Reactive Behaviour
    ticket Buy(RealWorldPlace departure, RealWorldPlace destination);
    price GetPrice(RealWorldPlace departure, RealWorldPlace destination);
  behaviour:

  ...
  implementation:
    ...
};

```

Figure 3: Travel Agency

agents. The operational semantics of an autonomous operation causes its invocation each time it is finished. Figure 2 extends the shopper agent with an autonomous operation *SearchTicket*, which models the task that is delegated to the shopper agent.

### 3 Traveling Agents

The discussion of the previous section adheres to a view of the agent world known as the computational view of CORRELATE: the agents reflect the natural abstractions of the application's problem domain. In this phase, hardware architecture aspects (like distribution) are completely hidden for the programmer, who can concentrate on modeling the various agents.

For example, Figure 2 shows that the agent is modeled without regard to distribution, target hardware or software platforms, etc. The computational model of CORRELATE essentially presents the programmer with an agent space that is homogeneous, even though the actual execution environment is not. The shopper invokes operations on *TravelingAgency* agents (outlined in Figure 3), their location is irrelevant as far as the semantics of the interaction itself is concerned.

Afterwards and separated from the first phase, an optimal execution must be obtained by instantiating an appropriate run time system for the application. Such a run time system should guarantee reliability, performance and security. The focus of this report is on performance and security.

The programmer works in the physical view, where all aspects that were previously hidden are now exposed. For our shopper agent, a logical operation would be to make distribution explicit and actually move the agent to the location of the *TravelingAgency*, if only to reduce communication overhead and thereby increase performance. In this model, the agent actually moves through a large heterogeneous network, traveling from host to host. For this purpose, the dynamic integration of CORRELATE agents in a host

environment is supported by the CORRELATE run time system.

### 3.1 Metaobjects in the CORRELATE Run Time System

CORRELATE and its run time system are based on a meta-level architecture. An architecture is called a meta-level architecture when meta-level objects are explicitly available for inspection (observation) and modification. In other words, a meta-level architecture enables the modification of the meta-system, possibly during the execution of the application. To support a meta-level architecture in an object-oriented design, one can define a metaobject protocol (hereafter MOP) that defines the set of metaobjects and their interactions (protocols). Specialized support systems can be built by specializing the predefined metaobjects.

In CORRELATE, a default metaobject treats a application object (agent) as an abstract process that is created, that sends and receives messages and that eventually is destroyed. In other words, the interface of a metaobject is application independent. Consequently, metaobjects are relatively easy to reuse.

In the context of this paper, two specific elements of the MOP have to be stressed:

1. All messages (incoming and outgoing) are intercepted by the metaobject. This is important for the development of agent specific location control objects.
2. Metaobjects are active objects. Metaobjects are programmed in the CORRELATE language framework, just like other (active) objects and they are treated in the same way<sup>2</sup>. Since a metaobject is just another active object, it runs concurrently with its base-level object.

The CORRELATE MOP is summarized by listing the header of a default metaobject in Figure 4.

Based on the two properties mentioned above, it becomes attractive to implement location control objects as specializations of a default metaobject. Figure 5 shows a first example; the metaobject will, each time an outgoing invocation is processed, migrate the agent to the host of the callee. The name server is consulted for this purpose<sup>3</sup>.

A second example is presented in Figure 6. Here the metaobject records all the interactions of the application object in a histogram. From time to time, the metaobject evaluates the object's location by executing an autonomous operation *MobilityPolicy*. In this case, we have assumed that the application agent will not migrate to the node of the callee on each interaction, but rather reside on the node where most of the interaction takes places. The example assumes that the method *IsEVALUATING* determines whether the embedded histogram has sufficiently changed.

### 3.2 The Kernel of the Object Support System

Both the computational part of the agent and its location control subsystem operate on top of a nucleus of the object support system, that offers:

---

<sup>2</sup>This means that they can have metaobjects as well if necessary.

<sup>3</sup>Note that invocation messages are reified: they encapsulate the identity of the sender and of the receiver of a message. This capability has been exploited in the code fragment of Figure 5.

```

active MetaObject {
autonomous:
    void Activate();
        // process one message from the incoming message queue
interface:
    void Construct(ConstructorMessage msg);
    void Delete(DestructorMessage msg);
    void MessageIn(InvocationMessage msg);
        // put in incoming message queue
    void AMessageOut(InvocationMessage msg);
        // forward to receiver
    void SMessageOut(InvocationMessage msg);
        // forward and become BLOCKED
    void End(InvocationMessage msg);
        // become READY
    void ReplyMessage(InvocationMessage msg);
        // accept and become RUNNING
behaviour:
    bool IsREADY();
    bool IsRUNNING();
    bool IsBLOCKED();
    for Activate() precondition IsREADY();
};

```

Figure 4: A default metaobject

```

active LocationControlMetaObject : public MetaObject {
interface:
    void AMessageOut(InvocationMessage msg) {
        Migrate(NameServer.LookUp(msg.GetReceiver()));
        MetaObject::AMessageOut(msg);
    }
    void SMessageOut(InvocationMessage msg) {
        Migrate(NameServer.LookUp(msg.GetReceiver()));
        MetaObject::SMessageOut(msg);
    }
};

```

Figure 5: An obvious metaobject for location control

```

active LocationControlMetaObject : public MetaObject {
autonomous:
    void MobilityPolicy(void);
interface:
    void MessageIn(InvocationMessage msg) {
        SB_interactionHistogram.Update(msg.GetSender(),msg.GetMethod());
        MetaObject::MessageIn(msg);
    }
    void AMessageOut(InvocationMessage msg) {
        RB_interactionHistogram.Update(msg.GetReceiver(),msg.GetMethod());
        MetaObject::AMessageOut(msg);
    }
    void SMessageOut(InvocationMessage msg) {
        RB_interactionHistogram.Update(msg.GetReceiver(),msg.GetMethod());
        MetaObject::SMessageOut(msg);
    }
behaviour:
    bool IsEVALUATING();
    for MobilityPolicy() precondition IsEVALUATING();
implementation:
    Histogram SB_interactionHistogram; //sender based information
    Histogram RB_interactionHistogram; //receiver based information
};

```

Figure 6: Location control can be based on interaction histograms

1. *Synchronization*: one operation is allowed to access the base-level object at a given time.
2. *Location transparent object interaction*: this is supported by introducing a remote reference and a distributed name service. A remote reference object packs all incoming invocations and forwards them over a communication network to the reference on another host.
3. *Agent-migration*: this is realized in three steps. First, a buffering reference is created: it blocks all incoming invocations. Then, the agent is moved to the destination address space where it is reinstalled with a new metaobject. The final step consist of replacing the buffering reference with a remote reference pointing to the new address space. A marshaling capability is provided by the framework: an object is packed into a stream of bytes for transport over the physical network. On the destination host, the object is reconstructed out of this stream.

This kernel is not the subject of this report. The work has been described in [1].

## 4 An approach to deal with security

A key concern for an agent run time system is security: the system will not only support known (local) agents, but it also has to deal with agents that originate elsewhere. It goes without saying that we do not allow opaque binaries to enter the local run time environment and provide them with processing resources. Malicious agents must be identified and access to local resources (such as disk space, files, processor time, etc.) must be denied. In our opinion, an interesting way in which a host environment can evaluate an agent and its intentions, is through the evaluation of its (high level) source code. This can be achieved by dynamically compiling or by interpreting this code.

It is true that source code evaluation is not the basis for absolute security. In our current perspective, we expect cases to appear in which the host environment is “not convinced” that the visitor is sound<sup>4</sup>. The basic point is that host environments can obtain maximal (but not necessarily sufficient) interpretation of the visitor’s goals when inspecting source code.

Of course, performance is a key issue in this context as well: the processing of source code is a source of overhead. Dependent on the characteristics of the application, one needs various solutions. We have developed a dynamic compilation facility and are in the process of building an interpreter. Dynamic compilation is in many cases an unacceptably slow process, especially when interactive behaviour of the application is dominant. The distinction between interpretation on the one hand, and dynamic compilation on the other hand, is based on the following observations:

- if the agent will only stay for a short while at the local node, its code is interpreted, yielding immediate response time. An example of this is the shopper agent from the previous sections.

---

<sup>4</sup>This can be compared with disallowing Perl scripts that include wild cards, a common practice on the Internet.

- if the agent is expected to reside for a substantial time at the local node, it might be worthwhile to invest some time in compiling the code, because after that it will execute faster. Agents that perform remote software upgrading (as discussed below) are an example here.

In both cases, the system inspects the code and refuses to support the agent if the latter contains potentially harmful instructions. This way, we can reconcile the performance demands with the security requirements.

### An Example

Remote software upgrading is an area where both dynamic compilation and interpretation are useful. Consider a set of traveling agencies that sell tickets following a certain policy. The policy is encapsulated in a *PricingPolicy* object which implements *GetPrice* and *SuggestPrice* operations through which users and client agents can negotiate about the price. If the policy changes, we like to charge an agent with the task of updating a set of *TravelAgency* agents with the new policy. The agent will visit each *TravelingAgency* in turn (logically or physically), check the pricing policy and, if applicable, update the component. The upgrader pays only short visits to each *TravelingAgency*, so its code is interpreted. The installed code however will remain in use for a potentially long time and will be used frequently, so it is compiled.

Figure 7 shows an upgradable travel agency that enables the dynamic modification of the pricing policy. Figure 8 shows the upgrader agent that visits a set of travel agencies<sup>5</sup>. The upgrader agent can update the *PricingPolicy* component by executing the language primitive *DynamicNew* to instantiate an object, hereby compiling it on the fly.

The default semantics of the *DynamicNew* primitive is that the object is created locally. This behaviour can however be changed when programming in the physical view, independently from the normal agent code. A local *DynamicLoader* object takes care of the actual compilation, verification and loading. In this example, we make sure the *PricingPolicy* object gets compiled at the site where it has to be installed, instead of at the location where the upgrader agent happens to be.

## 5 Related work

Research in the agent community can be roughly divided into two categories: work in distributed A.I. on how multiple agents can interact (e.g. in [3]) and work in distributed systems that support the execution of these agents in a distributed environment [4][5][7].

CORRELATE tries to bridge the gap between these two worlds by developing a concurrent object-oriented language with an open runtime environment. It is our view that this is the best solution for an agent language because concurrent objects offer expressive power and a natural approach to model the real world.

An important aspect of a COOL is the synchronisation of concurrent requests. Unlike other COOLs that address mobile agents (e.g. Obliq[2] and Telescript[6]), CORRELATE offers high-level language support for synchronisation. This minimizes the unwanted effects of the inheritance anomaly [8].

---

<sup>5</sup>Note that the *Upgrader* class exposes multiple autonomous operations.

```

active UpgradableTravelAgency{
  autonomous: // Autonomous behaviour
    void AnnouncePrices(){
      ...
      ... _aPricingPolicy$SuggestPrice();
      ...
    };
  interface: // Reactive Behaviour
    ticket Buy(RealWorldPlace departure, RealWorldPlace destination);
    price GetPrice(RealWorldPlace departure, RealWorldPlace destination){
      ...
      ..._aPricingPolicy$GetPrice(...);
      ...
    };
    PricingPolicy ReadPolicy(void);
    void SetPolicy(PricingPolicy*);
  behaviour:
  ...
  implementation:
    PricingPolicy* _aPricingPolicy;
    ...
};

```

Figure 7: A more flexible Travel Agency

```

active Upgrader{
autonomous: // Autonomous behaviour
    void InstallNewVersion(){
        if (OldVersion(_current_agency$ReadPolicy()){
            PricingPolicy* new_policy = DynamicNew(_new_code);
            if (new_policy) _current_agency$SetPolicy(new_policy);
        }
    };
    void Move();
    void ReturnHome();
interface: // Reactive Behaviour
    ...
behaviour:
    bool IsAtNewAgency();
    bool IsJobDone();
    bool IsBeenEverywhere();
    for InstallNewVersion() precondition IsAtNewAgency();
    for ReturnHome() precondition IsBeenEverywhere();
    for Move() precondition !IsJobDone() && !IsBeenEverywhere();
implementation:
    SourceDescription _new_code;
    LogicalPlace _current_agency;
    ...
};

```

Figure 8: An upgrading agent

The concept of an autonomous operation is somewhat similar to the "live" operation in Telescript[6]. Unlike in CORRELATE however, only one such operation is possible for each agent and the scheduling of the operation is under control of the execution engine. CORRELATE offers an open run time system, which makes sure programmers keep total control over the execution of their agents [9], without compromising the high level language support that is essential from a software engineering point of view.

Almost all agent languages we know, with Obliq being a notable exception, provide a very explicit view on distribution. CORRELATE on the other hand, separates the computational view with location transparent interaction from the physical view where the distribution is made visible.

## 6 Summary

In this report, we have presented two aspects of the concurrent object-oriented language CORRELATE. Firstly, we have introduced its computational view that offers programmers an agent space that is homogeneous with regard to distribution, security policy, etc. and in which different agents can be naturally modeled. Secondly, we have discussed the run time system with its metaobject protocol and its facilities for source code processing. Two important requirements have been addressed: (1) sufficient performance can be obtained by traveling agents because they can use an appropriate subsystem for location control and (2) the host system can in principle be protected from potentially malicious agents.

The CORRELATE prototype currently runs on different platforms: (BSD based) OSF/1, Silicon Graphics IRIX 5.3 and (SystemV based) Sun Solaris 2.x.

## Acknowledgement

The authors would like to thank Bart Vanhaute for his work on the dynamic compilation facility.

## References

- [1] Stijn Bijnens, Wouter Joosen, and Pierre Verbaeten. A Reflective Invocation Scheme to realise Advanced Object Management. In *ECOOP'93 Workshop on Object-Based Distributed Programming*, pages 142–154. Lecture Notes in Computer Science Vol. 791, Springer Verlag, 1994.
- [2] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.
- [3] Yves Demazeau and Jean-Pierre Mller. *Decentralized A.I.* Elsevier, July 1990.
- [4] James Gosling and Henry McGilton. The java language environment, a white paper. Technical report, 1995.

- [5] Robert S. Gray. Agent tcl: A transportable agent system. 1995.
- [6] General Magic Inc. The telescript language reference. Technical report, 1995.
- [7] G. Di Marzo and, M. Muhugusa, and C. Tschudin. The messenger paradigm and its impact on distributed systems. In *Workshop on Intelligent Computer Communications, ICC'95*, 1995.
- [8] Satoshi Matsuoka and Akinori Yonezawa. Analysis of the Inheritance Anomaly in Object-Oriented Concurrent Programming. In *Research Directions in Concurrent Object Oriented Programming*. The MIT Press, 1993.
- [9] Johan Van Oeyen, Stijn Bijmens, Wouter Joosen, Bert Robben, Frank Matthijs, and Pierre Verbaeten. A Flexible Object Support System as Runtime for Concurrent Object-Oriented Languages. In Chris Zimmermann, editor, *Metaobject Protocols*, chapter 12. CRC Inc., 1996.