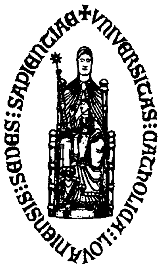
Building a Meta-Level Architecture for Distributed Applications

Bert Robben, Wouter Joosen, Frank Matthijs

Bart Vanhaute, Pierre Verbaeten

Report CW 265, May 1998..



Katholieke Universiteit Leuven

Department of Computer Science

Celestijnenlaan 200A - B-3001 Heverlee (Belgium)

Building a Meta-Level Architecture for Distributed Applications

Bert Robben, Wouter Joosen, Frank Matthijs

Bart Vanhaute, Pierre Verbaeten

Report CW 265, May 1998..

Department of Computer Science, K.U.Leuven

Abstract

Distributed applications are complex software systems that inherently expose concurrency and that need support for non-functional requirements such as reliability and security. We present a model that integrates application objects with a support architecture. An implicit concurrent meta-object protocol guarantees a clear separation between application and support system. Both are programmed with Correlate, a concurrent object-oriented language. We illustrate how a generic meta-level architecture can be customised to take application specific behaviour into account.

Keywords : object-orientation, distribution, meta-object protocol.

Building a Meta-Level Architecture for Distributed Applications

Bert Robben[1], Wouter Joosen, Frank Matthijs, Bart Vanhaute, Pierre Verbaeten
Dept. of Computer Science., K.U.Leuven
Celestijnenlaan 200A, B3001 Leuven - Belgium
Email: bert@cs.kuleuven.ac.be

Abstract

Distributed applications are complex software systems that inherently expose concurrency and that need support for non-functional requirements such as reliability and security. We present a model that integrates application objects with a support architecture. An implicit concurrent meta-object protocol guarantees a clear separation between application and support system. Both are programmed with Correlate, a concurrent object-oriented language. We illustrate how a generic meta-level architecture can be customised to take application specific behaviour into account.

1 Introduction

Meta-level architectures has become an important research topic in object-oriented programming. The area is closely related to reflective programming. Both reflective programming and meta-level architectures share the common goal of enabling software developers to write application programs that can somehow manipulate the state of their own execution[11].

The words *"state of execution"* can refer both to aspects of language implementation (for instance method lookup) as well as to elements of the system software (i.e. memory management). These two targets have most often been addressed in different and complementary research projects. This way, customised language implementations and execution environments have been developed over the last ten years.

A central theme in this research is the study of Meta-Object Protocols (MOPs hereafter). For a given application (i.e. application objects), the state of execution is reified in a set of meta-objects. The MOP is the documented interface to these meta-objects. A meta-level architecture (i.e. meta-level objects) uses the MOP to adapt and tune the behaviour of the application independently of its functionalities[12].

One particularly relevant problem is the reuse and the combination of existing customisations. This problem has been addressed in [10] where a model for the composition of meta-level objects is proposed. Our approach to the problem is complementary to this work in the sense that we aim at a meta-level architecture in which customisations can coexist without affecting each other. One could say that this approach is based on architecture, rather than on composition.

Our particular experience is related to the development of distributed applications with customised execution environments. Concurrency is inherent to distributed applications and we have developed a concurrent object-oriented language, Correlate, to support concurrency appropriately. The MOP has been used to build meta-level architectures that both model language aspects (i.e. the concurrency issues) as well as support non-functional requirements such as secure distribution, reliability, etc.

---

[1]Research assistant of the Fonds voor Wetenschappelijk Onderzoek - Vlaanderen (F.W.O.)

The rest of this paper is structured as follows. Section 2 covers some background on the development of distributed application software. Section 3 characterises Correlate, the concurrent object-oriented language that is the nucleus of our prototypes. Section 4 defines our MOP, which is a concurrent and implicit MOP that defines the interaction between base-level objects and meta-level objects. Section 5 describes how to build a meta-level architecture. Here we define an inter-meta-object protocol (orthogonal to our MOP) and illustrate how the overall architecture enables customisations, as well the combination of customisations. Section 6 gives an overview of related work. We conclude in section 7.

2 Development Model

The development of distributed applications is inherently difficult. Hence, a clean and powerful program development process is required to achieve reliable and efficient software that remains (trans)portable, maintainable and extensible. Our approach is based on two main themes: separation of concerns and concurrent objects.

2.1 Separation of concerns

In our approach, the top down development of distributed applications involves two phases. First a computational model is developed which only includes abstractions from the application's problem domain. In this phase, distribution aspects are completely hidden for the application programmer, who models an application as a set of autonomous objects that operate concurrently. Secondly, a physical model must be developed to target the application to the specific architecture it is running on. In this model typical non-functional requirements like reliability, performance and security are realized.

The separation between these two models is guaranteed by an implicit concurrent meta-object protocol (MOP). This MOP makes abstraction of the application specific details of the computational model and offers the meta-level a generic model of a base-level application. We have chosen for a strictly implicit meta-object protocol. No combination with an explicit protocol is allowed: a base-level object cannot directly call meta-level objects. The details of our MOP are explained in detail in Section 4.

Meta-level objects are introduced that use the MOP to effectively control the application objects. They realize mechanisms like object migration and object replication. Meta-level objects can also appear as full-fledged autonomous subsystems that implement a certain policy on top of the mechanisms offered by other meta-level objects. A load balancing system on top of a mobility mechanism is an example. A system of interacting meta-level objects that realizes a non-functional requirement is called a *meta-level architecture*.

The abstract interface between base-level application and meta-level architecture makes it possible to reuse a meta-level architecture for a set of different applications. Alternatively, different runs of one application can also be controlled by different meta-level architectures. For instance, the application is first tested with a debugging meta-level architecture. In a second phase, it can be distributed with an appropriate distribution meta-level architecture.

A potential weakness of such a separated approach is that it can lead to an overly systematic unification where abstraction is made of all differences between application objects. We show further on a variety of techniques to make a meta-level architecture take application specific characteristics into account.

2.2 Concurrent objects

Distribution inherently exposes concurrency. The base-level application is modelled as a set of active autonomous objects. For the meta-level architecture, we essentially apply the same approach. This has many benefits. Often distributed algorithms like election or reliable broadcast are described in terms of a set of concurrent processes. This naturally maps on active objects. Meta-level architectures exploit the expressive power of active autonomous objects. An autonomous meta-level architecture monitors an application's behaviour and takes the initiative to adapt the base-level application's behaviour. Take for instance an autonomous dynamic load-balancing subsystem that ensures a reasonable workload distribution in a networked environment. The load balancers typically distribute information about the local workload. When a node becomes over loaded, the autonomous load balancer will act and migrate some objects to other locations based on the load distribution information that is locally available. The entire load balancing policy is encapsulated in the load balancing subsystem.

In a classical approach, the support system is built as a passive system that offers an interface to the application. All initiative will be situated in the base-level application where application objects will directly call the subsystem when adaptation is required. The subsystem still implements the mechanisms, but some parts of the policy end up inside the application. A major drawback of this approach is that the coupling between base-object and meta-level object becomes stronger and it will be much harder to integrate the application with different subsystems in the meta-level architecture.

To enable a high-level model of autonomous active objects, we offer the programmer the high-level programming language Correlate that integrates the concepts of concurrency within a sequential object-oriented programming language. Instead of creating a new language from scratch, the approach we take is to enhance an existing object-oriented language with a set of constructs to handle concurrency. Currently, we have prototypes running on top of C++ and on top of Java.

3 The Correlate Language

This section summarizes the Correlate language. A complete description and formal semantics can be found in [7].

3.1 Active objects

Correlate provides the application programmer with a model of *active objects*. Active objects control the synchronization of concurrent requests. In Correlate, active objects are units of concurrency; only one operation is allowed to execute on an object at the same time. 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 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. In Correlate, this issue is expressed with synchronisation constraints. Each operation can be annotated with a precondition that evaluates whether the object is ready to accept the operation in its current state or not.

To maintain encapsulation as much as possible and to reduce the effects of the inheritance anomaly, Correlate objects 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 internal operations that describe the abstract states as simple functions on the state of the object, i.e. no object interaction or side effects are allowed. Preconditions are specified as expressions of

these abstract states and the parameters of the invocation.

3.2 Autonomy

In the Correlate object model an object can exhibit *autonomous behaviour*. This way an object can perform a spontaneous action which is not triggered by external stimuli (incoming messages).

At the level of a class definition, the language's syntax makes a distinction between autonomous operations and interface operations. Interface operations represent the reactive behaviour and are conceptually the same as public operations in object-oriented languages like Java or Smalltalk. *Autonomous* operations are invoked as soon as the object has been created and they are re-invoked when the execution of the operation is terminated.

3.3 Object Interaction

Objects interact by sending messages (invoking interface operations). Different models for message passing exist. Synchronous message passing blocks the sender's activity until the operation is executed to completion in the receiver object. With asynchronous message passing, the sender does not have to wait for the completion of the invocation, thus increasing concurrency. Correlate supports both *synchronous* and *asynchronous* message passing. The application programmer must explicitly specify[2] what kind of message passing (s)he prefers.

3.4 Lifetime of an object

Correlate objects are created dynamically with the **new** operator (just like in Java or C++). Once the constructor of an object is executed, it can start to accept messages (both interface and autonomous). Correlate objects are not garbage collected automatically (as it is very difficult to determine when an autonomous entity becomes garbage). Explicit destruction is supported through a **delete** operator. As with message passing, the application programmer must specify whether the creation (destruction) happens synchronously or asynchronously.

3.5 Syntax

The following code fragment gives a glimpse of the language syntax. The basic idea is to enhance a host language (in this case Java) with some additional keywords and constructs that express the new concepts.

```
active class Controller {
  autonomous protected void check() precondition isOn() {
    if (tooWarm) heater $ stopHeating();
    if (tooCold) heater $ startHeating();
  }
  public void turnOff() { ... }
  public void regulateTemperature(Temp tmin, Temp tmax) {
    heater = new @ Heater();
  }
  internal boolean isOn() { ... }
  protected Heater heater;
  ...
}
```

---

[2]A synchronous invocation is annotated with a '$' character, an asynchronous one with '@'.

The example shows a `Controller` class that implements functionality to maintain the temperature in a certain range. The autonomous operation `check` monitors the environment and adjusts the temperature by interacting with a heater object. When the `turnOff` operation is called, the `Controller` shuts down and stops monitoring.

4 A Concurrent MOP

Our meta-object protocol guarantees a separation between the computational and the physical model of an application. It abstracts away from the application specific details and offers the meta-level architecture a generic model of an application. We have chosen for an implicit meta-object protocol: direct interaction of a base-level object with its meta-level architecture is not allowed.

4.1 Meta-level representation of base-level interactions

To build distributed meta-level architectures, control is required over interactions between base-level objects. Four different kinds of object interaction are supported in our MOP: method invocation, object construction, object destruction and autonomous behaviour. Each is reified in a specialized `Message` object. The return value of an executed operation is reified in a `ReplyMessage`.

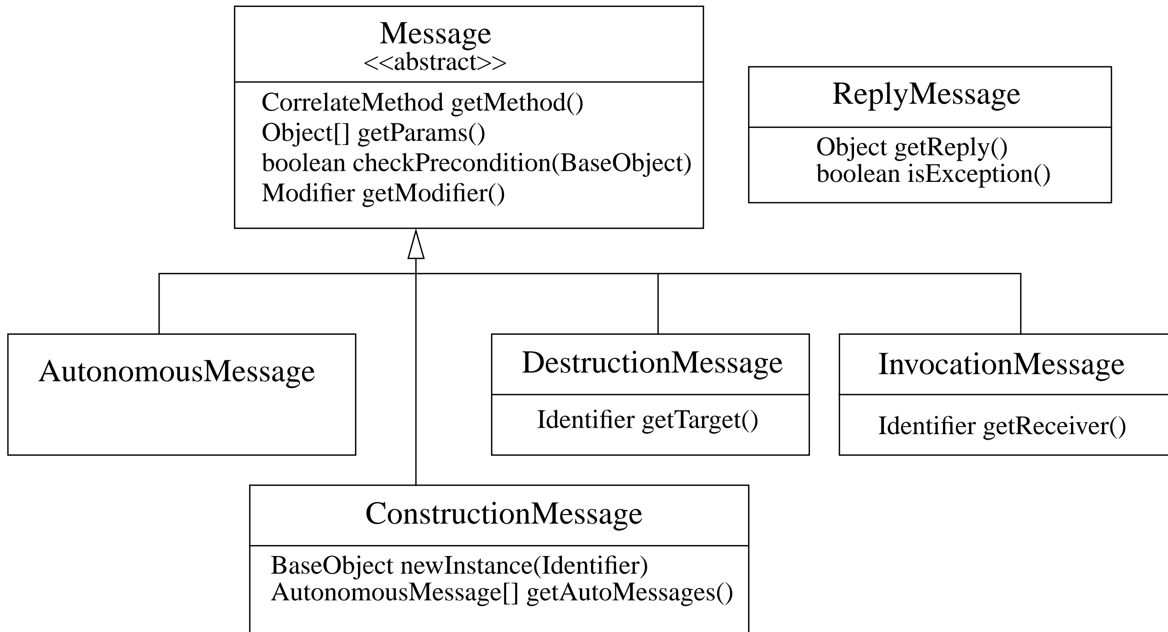


Figure 1: Message class diagram

Figure 1 shows the class diagram of the reified messages. The class `Message` contains selectors that allow introspection of the operation that is called and of the invocation parameters. The `Modifier` object represents additional information concerning the invocation. In its most basic form, it merely specifies whether the interaction is synchronous or asynchronous. The `checkPrecondition` operation evaluates the synchronization constraints of the operation and indicates whether the invocation can be executed in the current state or not. The class `ReplyMessage` is a simple class that contains the result of an invocation. The `isException` operation indicates whether the result is an actual return value or rather an exception that has been thrown while executing the operation.

4.2 Meta-level representation of base-level objects

An object has state, behaviour and identity and these properties are represented at the meta-level by `BaseObject` objects, `Activation` objects and `Identifier` objects.

State reification

By default, only coarse-grained access is provided to write/read the entire object to/from a stream.

```
public class BaseObject implements Serializable;
```

However, a meta-level object sometimes needs more fine-grained information on the state of the base-level object. An example is object synchronization. The meta-level architecture implements the concurrency constraints on the base-level application but in order to do so, it needs information about the state of the base-level objects to decide whether or not a certain operation is allowed. In our experience, this kind of state dependent behaviour is virtually impossible to realize strictly at the meta-level without a cumbersome protocol to replicate the (partial) state of the base-object at the meta-level. A simple synchronization problem can serve as an example. Take e.g. the well-known case of a bounded buffer where `put` (resp. `get`) operations are not allowed when the buffer is full (resp. empty). To implement this synchronization policy only the current size of the buffer needs to be available. This information can be maintained easily at the meta-level. If the buffer becomes a set however, this is no longer true. If the meta-level is to maintain the size of a set, it needs to keep track of all elements in it, thereby mirroring almost the whole base-object's state and behaviour.

Providing direct access to the base-level object's internal state is not a solution. This clearly breaks the encapsulation principle and will cause a lot of trouble in case of specialization. State abstraction functions are introduced as a simple solution. These are functions that compute without side-effects a value dependent on the state of the object. Preconditions that model synchronisation constraints are clear examples. Other more specific abstraction functions can be defined by the application programmer through internal interfaces. An example is given in Section 5.

Behaviour reification

To control concurrency, ongoing base-level computations need to be reified. One approach would be to create evaluator objects at the meta-level that interpret the base-level code. These evaluators give the meta-level very fine grained control up to the level of single statements and expressions. We take a more coarse grained approach. Computations are reified in `Activation` objects that expose the parts of the state that are required for object interaction. The non-reified parts are compiled to a more efficient format.

```
public class Activation {
  void start(Message msg, BaseObject bo);
  boolean hasFinished();
  ReplyMessage getReply();
  boolean isSuspended();
  Message getMessage();
  void proceed(ReplyMessage msg);
}
```

Figure 2 shows the state diagram of an `Activation`. The `start` operation brings an `Activation` in state RUNNING and starts the computation on the base-object asynchronously.

When the base-object interacts, the state changes to SUSPENDED. The `Activation` object can then be queried which `Message` it wants to send. It resumes execution after the `proceed` operation is called with the respective reply message. In case of asynchronous interaction, the value of this parameter can be `null`. When the base-level computation ends, the state shifts to FINISHED and the reply value can be fetched.

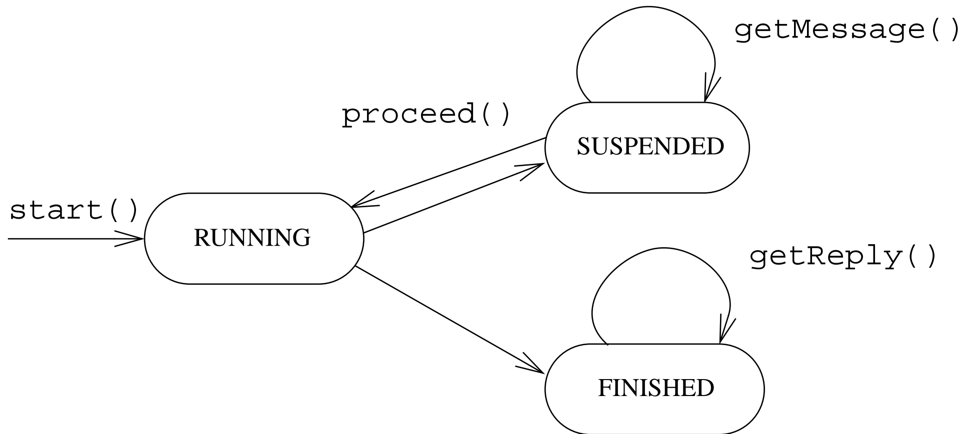


Figure 2: Activation state diagram

Base-level and meta-level run concurrently as activations are always started and resumed asynchronously. For the `BaseObject`, no concurrency control at all is provided. In principle, there is no constraint on the number of concurrent `Activation` objects that can be active on a single base-level object. A meta-level architecture that implements the default Correlate semantics (prohibiting intra-object concurrency) requires only one activation per base-level object. To ensure that application specific semantics are not violated, the `checkPrecondition` operation on class `Message` should be used to determine which `Message` objects are acceptable in the current state for a given base-level object.

Identity reification

If the meta-level architecture wants to absorb various policies to support distribution and persistence, tight control is required over the boundary of an object. This is realized by introducing `Identifier` objects.

```
public interface Identifier;
```

These objects represent the identity of a base-level object. Base-level objects refer to each other using such `Identifiers`. It is the task of the meta-level architecture to implement the mapping from `Identifier` object to base-level object. This concept of `Identifiers` effectively makes it possible to isolate a base-level object from all others. This is clearly necessary e.g. in case meta-level objects want to migrate a single application object to another host or when meta-level objects want to swap the application object out to disk.

4.3 The meta-tower

A meta-level architecture consists of objects written in Correlate. This has the benefit that they can exploit the high level features of the Correlate language. Another consequence is that a meta-level architecture can be built on top of them as well. The same reasoning can be applied to this meta-meta level architecture giving rise to a potentially unlimited tower of meta-level architectures. As an example, a meta-level architecture can provide reliability in the form of a check-

pointing protocol. Another meta-meta level architecture on top of it can take care of secure distribution. A drawback of multiple levels is increased complexity. To understand what happens with an invocation at some level, you have to know the working of all meta-levels above it. When the tower becomes dynamic, i.e. can change at run-time, this situation becomes even more complex.

An important problem that needs to be addressed is the building of a finite implementation of this possibly infinite tower. For an explicit MOP this is a serious problem as each object can always interact with its meta-object. Existing systems use lazy instantiation mechanisms or limit the number of meta-levels. For an implicit MOP, the solution turns out be fairly straightforward. Because an object can never interact directly with its meta-level architecture, the language run-time[3] never needs to instantiate a default meta-level architecture. Dynamic extension of the meta-tower only happens in a constrained manner: the tower only grows when a base-level application becomes a meta-level architecture and starts managing another application through the MOP defined in this section. The resulting development model reflects this by requiring that the decision of which meta-level architecture to attach to a given base-level application needs to be made before the application starts. Once it is running, its meta-tower can only change insofar as the meta-level architecture has been explicitly designed to change; the meta-object protocol provides no support whatsoever.

5 Building a Meta-Level Architecture

This section explores the physical model in more detail. First we sketch the basics of a simple meta-level architecture that supports distributed mobile objects. This shows the expressive power of our proposed MOP. A second part explores how a meta-level architecture can be constructed to take application specific requirements into account.

5.1 A default meta-level architecture

This section shows a possible meta-level architecture that implements the default Correlate semantics. The class `MetaObject` is defined. An instance of this class is a meta-level representation of one base-level object. `MetaObject` aggregates the basic building blocks defined in the previous section. An important next step is to define the external interface that `MetaObject` instances use to interact.

```
active public interface MetaObjectInterface {
  void request(Message msg, Identifier sender);
  void reply(ReplyMessage msg);
}
```

The `request` operation models the server-side. Clients can call it to request for an invocation on or for the destruction of the base-level object. The `reply` operation represents the client-

[3]The *language run-time* is defined as the black box that implements the meta-object protocol and that supports the run-time execution of an application written in Correlate.

side and is used to return the reply[4] of a requested synchronous interaction.

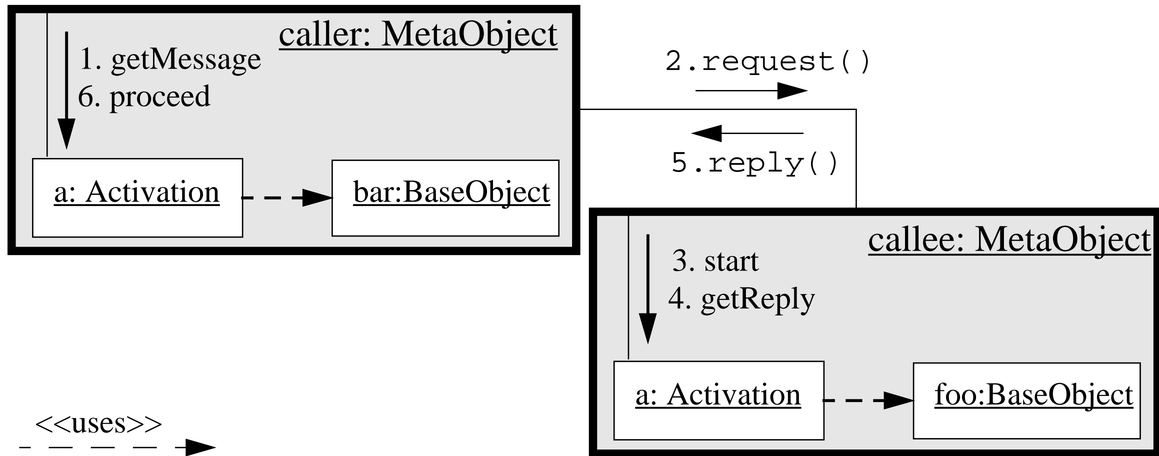


Figure 3: Scenario diagram

Figure 3 shows a simple scenario where an object of class `MetaObject` starts a computation on its base-object. During this computation, a synchronous invocation is made by base-object *bar* on base-object *foo*. *Caller* and *callee* use the `MetaObjectInterface` to transfer the request and the reply. Objects are created in a similar way, but a new `MetaObject` is created and its constructor called, instead of invoking the `request()` operation on an existing `MetaObject`. In the case of asynchronous interaction, the `reply()` operation is not used and the caller's activation can proceed immediately.

The following code fragment shows the `MetaObject` class. Two autonomous operations implement the monitoring of the state of the current `Activation`. Requests are only accepted when the base-object is not busy (i.e. no activation is running) and when the synchronization constraints of the operation allow it.

```
public active class MetaObject implements MetaObjectInterface{
  MetaObject(ConstructionMessage msg,
    Identifier myId, Identifier sender) { ... }
  autonomous void handleInteraction()
    precondition isBusy() && myAct.needsInteraction() { ... }
  autonomous void endOfActivation()
    precondition isBusy() && myAct.hasEnded() { ... }
  public void request(Message msg, Identifier sender)
    precondition isReadyToAccept(msg) { ... }
  public void reply(ReplyMessage msg) { ... }
  void handleAutoMessage(AutonomousMessage msg)
    precondition isReadyToAccept(msg) { ... }
  internal boolean isReadyToAccept(Message msg) { ... }
  internal boolean isBusy() { ... }

  protected Activation myAct;
  protected BaseObject baseObject;
  protected Identifier myIdentifier;
}
```

[4]The result of the destructor/constructor is an acknowledgement that the object is actually destroyed/created.

5.2 Building support mechanisms

This simple `MetaObject` is extended to create a meta-level architecture that deals with distribution. Two new key classes are introduced, `LocalMetaObject` and `RemoteMetaObject`. The former is an extended `DefaultMetaObject`, it represents a local base-level object. The latter is a meta-level object that represent a base-level object that resides on another host. A `RemoteMetaObject` object always forwards requests and replies to that other host. The `Host` class encapsulates the lower-level mechanisms of inter-host communication.

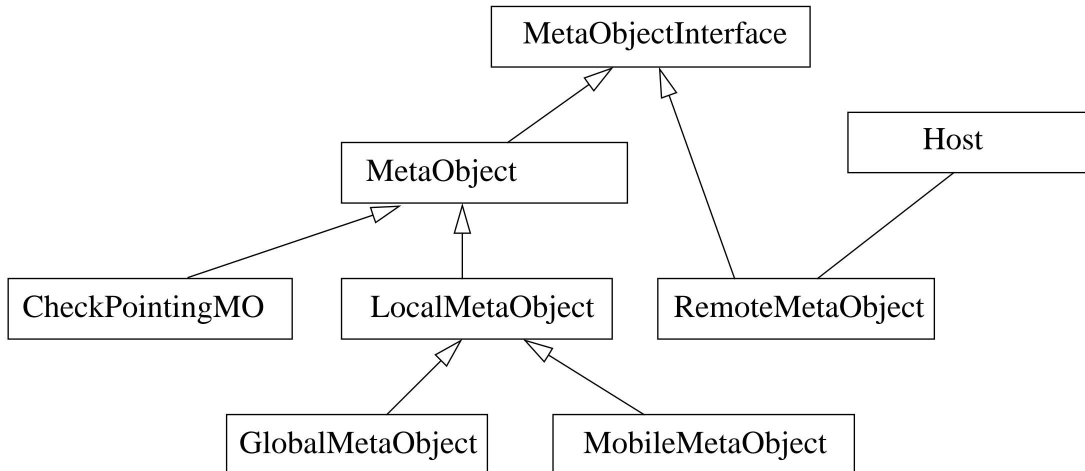


Figure 4: MetaObject diagram

These two classes represent only the basic mechanisms. Certain applications have more specific needs than location indepent object invocation. One example is a search agent that roams the network in search for interesting information. A vital aspect of these kinds of objects is their ability to migrate from one host to another. That way, a lot of communication bandwidth can be gained by moving the object to the locations of the information sources.

The class `MobileMetaObject` implements the object migration mechanism[5]. It combines in fact the functionality of a `LocalMetaObject` and a `RemoteMetaObject`. When the base-object is local, it behaves exactly like the former and vice versa in case it is remote. The interesting bit is the implementation of the state change; i.e. what happens when the base-object moves. Figure 5 shows the situation where an application object moves from lucky-luke to joe. Three different scenarios are possible. The easiest one takes place at host jack and is when a remote object moves to another remote host. Only a simple (even lazy) update of the host link is required. In the other two scenarios, the base-level object will be respectively sent and received, which is no big deal as a `BaseObject` permits serialization. `Activation` objects are not seri-

[5]O*bject* migration concerns issues of transferring the state of an object. This is not to be mistaken by *code* migration which is a complimentary mechanism that transfers classes (i.e. code).

alizable: migration is thus only possible when none are running on the base-level object.

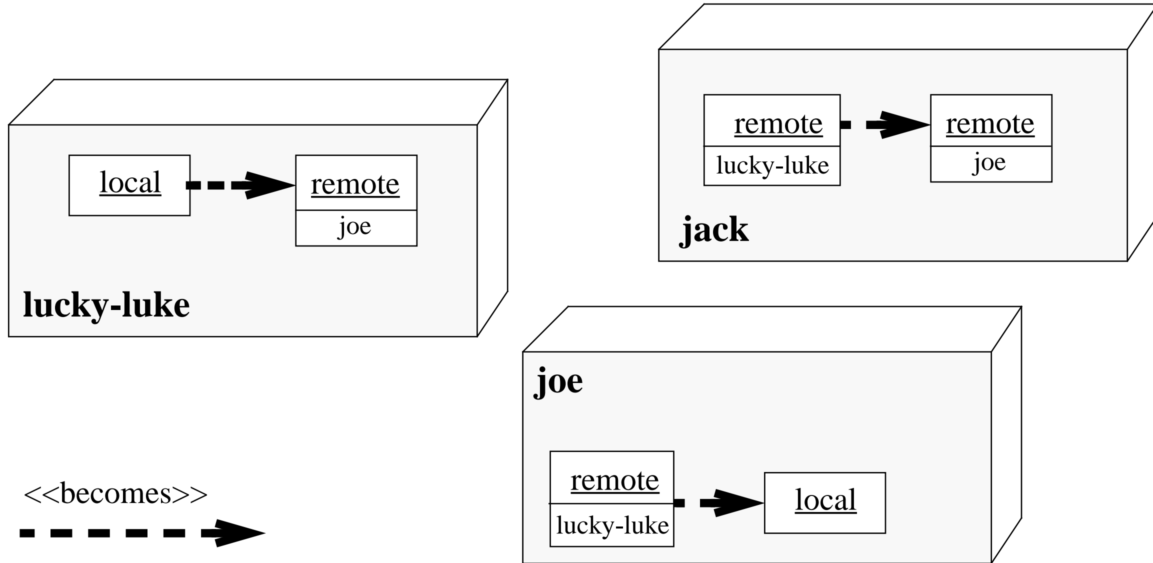


Figure 5: Object Migration

The following code fragment describes the process in more detail. `MigrateMessage` and `UpdateMessage` are simple objects that, once transferred to the destination host, invoke the `migrate` and `update` operation on the meta-object associated with the given `Identifier` object.

```
public void move(Host newHost)
precondition !isBusy() {
  newHost@send(new MigrateMessage(baseObject, myIdentifier));
  baseObject = null;
  state = REMOTE;
  host = newHost;
  network @ broadcast(new UpdateMessage(newHost, myIdentifier));
}
public void migrate(BaseObject bo) {
  state = LOCAL;
  baseObject = bo;
}
public void update(Host newHost) {
  host = newHost;
}
```

A second example of an often useful mechanism is provided by a `GlobalMetaObject`. The idea here is that some applications make use of large global objects that contain static data, i.e. data that never changes. A dictionary for a translation program or a distance matrix for a Traveling Salesman solver are examples. When these applications are distributed, a trivial consistency protocol can be used to replicate these objects over multiple hosts. The `GlobalMetaObject` class is a simple extension of `LocalMetaObject` that implements such a policy.

A final, more complex example, is a meta-level architecture that encapsulates a checkpointing algorithm. This meta-level takes persistent snapshots of a running application to prevent the

loss of all information in case of an occurring failure. [9] describes such an algorithm that determines a global state for a fixed set of communicating processes. The algorithm defines a process as an entity that has a state and can send and receive messages. Both messages and state need to be serializable. No other characteristics of a process are required for this algorithm. These requirements are met by the MOP and a `CheckPointingMetaObject` has been developed.

5.3 Fitting a meta-level architecture to a base-level application

The mechanisms of the previous subsection establish the basis to build meta-level architectures that realize non-functional requirements. This subsection explores to what extent we can keep and want to keep these architectures independent of the base-level applications.

Orthogonal meta

The easiest case is where the meta-level is completely orthogonal with respect to the application. From the point of view of the meta-objects, all base-objects behave according to some the same policy. There is no intrinsic difference between one base-object and another; all are treated equally. An example is the use of checkpointing meta-objects. Base-objects are considered to be autonomous active objects that interact asynchronously[6]. The checkpointing meta-level will accept any such application and make it fault-tolerant.

State dependent behaviour

In a more complex example, the meta-level architecture needs information about the state of individual application objects. Revisit the example of a dynamic load balancer that distributes an application across a cluster of workstations connected through a local network. The load balancer objects observe the local load and try to distribute the application objects evenly by migrating them from overloaded hosts to less heavily used machines. However, not all application objects can be migrated at all possible times. For instance, objects that temporarily use local resources like a display or the local file system cannot always be migrated to another machine. If the load balancer is to take this into account, it needs to know more about the state of the application objects. A simple internal interface `Mobile` is defined. Application classes can implement this interface to provide the information to the meta-level.

```
internal interface Mobile {
  boolean readyToMove();
}

public void move(Host newHost)
precondition ((Mobile) baseObject).readyToMove() && !isBusy();
```

An improved `MobileMetaObject` could e.g. use this information in the precondition of its `move` operation to delay a migration request until the application object is ready to go.

Application specific meta-level classes

Finally, the meta-level programmer can design his meta-level architecture as a framework. This allows the application programmer to plug in certain classes to get very application specific behaviour. This does not have to be a very hard thing to do as the same paradigm is used for meta-level and application. A simple example can be taken from the distribution meta-level. When an application object creates a new object, its meta-object needs to decide where (i.e. on

---

[6] This restriction is a consequence of the not-serializability of `Activation` objects.

which host) it is to be allocated. Application specific knowledge about the behaviour of the new object can certainly play an important role here to make a reasonable choice.

The meta-level programmer can support this by introducing an abstract class `Allocator`. This class has one operation that computes the optimal location to create the new object with the given `ConstructionMessage`. A `DefaultAllocator` is created that always determines the local host. An application programmer can however write his/her own implementation of this class to define a specific allocation policy. Figure 6 shows the class diagram.

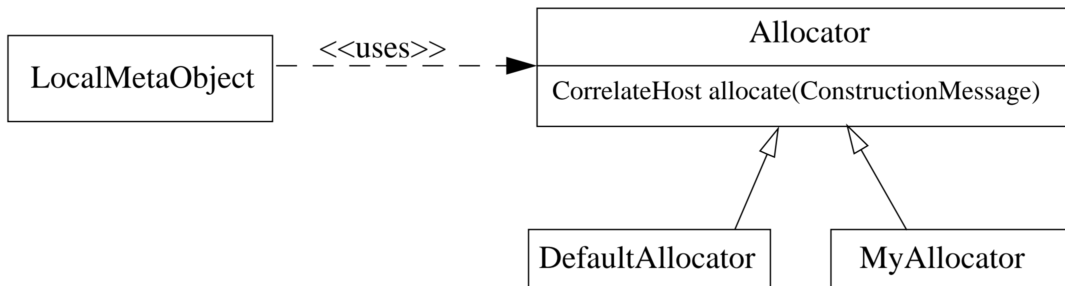


Figure 6: Distributed Object Allocation

This technique can of course be combined with the previous one to create very sophisticated application specific behaviour.

5.4 Building a meta-tower

A meta-level architecture can be a complex piece of distributed software. For example, take an electronic diary application that needs to be made persistent and distributed. A straightforward approach would create one meta-level architecture that supports the combined functionality. An alternative and more flexible approach is to apply the full development model. First, a subsystem is built that takes care of persistence. In a second phase, this application (which is in fact already a meta-level architecture) is distributed by adding a distribution meta-level.

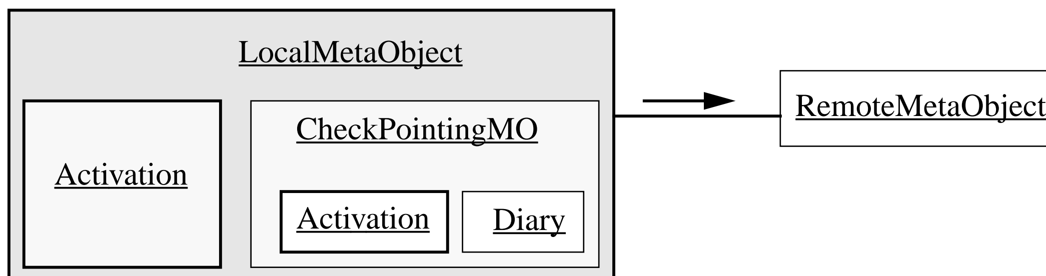


Figure 7: Distributed Checkpointing

6 Related work

A lot of work in reflection and meta-level architectures has been performed. The CLOS MOP e.g. provides an extensible implementation of CLOS: all specifications of CLOS are modifiable. The aim of our meta-level architecture is more restricted and focuses on problems related to concurrency and distribution.

ABCL/R[5] is a reflective concurrent interpreted programming language. It includes a very powerful explicit meta-object protocol where an object's state, scripts (code) and script evaluator are reified. Correlate is a compiled language and our MOP reifies only a fraction of an object's

behaviour to reduce the costs associated with the meta-level architecture.

Open C++[2] is similar to our work in that it supports an implicit meta-object protocol for a compiled language (C++). However it is based on a sequential object model and as such not ideally suited for distributed applications. Also, the identity of a base-level object is not reified, making it hard to control references that base-objects have to each other.

The CodA[8] meta-architecture explicits supports active objects and factors the meta-level into objects. An important difference with our work is that in CodA the objects at the meta-level are sequential.

The AL-1/D[4] project shares many aspects with our work. It is based on concurrent objects and uses a reflective architecture to support distributed applications. It proposes a Multi Model Reflection framework were a base-level object is under control of a set of six meta-objects each with a well-defined task and protocol. The six default meta-object implement the distributed object model of AL-1/D[3]. Meta-level programmers can extend these default meta-objects to create their own specific policies. Influenced by RbCl[6], an important design decision in our work is to keep the run-time kernel very small to allow the meta-level programmer as much freedom as possible. A large difference can be found at the language level. Where Correlate provides high-level support for concurrency, AL-1/D merely presents, to our knowledge, a very simple language framework.

7 Conclusion

Distributed applications are complex software systems that inherently expose concurrency and that need support for non-functional requirements such as reliability and security. This paper presents an implicit meta-object protocol for the concurrent object-oriented language Correlate. Correlate extends an existing sequential language to offer support for concurrency. Current prototypes are integrated with Java and C++. The MOP reifies object interaction and enables the construction of mechanisms like object migration and object replication. It is shown how these mechanisms can be used to create a meta-level architecture that realises the non-functional requirements. The implicit MOP guarantees a clear separation of concerns between application and support subsystems and makes it possible to reuse a meta-level architecture for a set of different applications.

Our contribution lies in the emphasis on concurrency: both application and support system are modelled as a set of autonomous active objects. We show how a complex support system can be built as a meta-level architecture and how it can be specialized to take application specific elements into account. Our experience with the prototype indicates that a diverse approach like ours is very useful to build concurrent and distributed software. Both algorithms and language support as well as reflection are necessary and complementary aspects to built distributed systems.

Acknowledgements

We would like to thank Stijn Bijnens and Johan Van Oeyen for many fruitful discussion over the last couple of years; these are at the base of many ideas in this work. We also wish to thank Eddy Truyen for his work on applying the MOP in the domain of fault-tolerance.

References

[1] Johan Van Oeyen, Stijn Bijnens, Wouter Joosen, Bert Robben, Frank Matthijs and Pierre Verbaeten. “A Flexible Object Support System as Run-time for Concurrent Object-Ori-

ented Languages". In Chris Zimmermann, editor, *Metaobject Protocols*, chapter 12, CRC Inc, May 1996.

[2] Shigeru Chiba and Takashi Masuda. "Designing an Extensible Distributed Language with a Meta-Level Architecture". In *Proceedings ECOOP '93*, pages 483-502, Kaiserslautern, July 1993. Springer-Verlag.

[3] Hideaki Okamura and Yutaka Ishikawa. "Object Location Control Using Meta-level Programming". In *Proceedings ECOOP '94*, pages 299-319, Bologna, July 1994.

[4] Hideaki Okamura, Yutaka Ishikawa and Mario Tokoro. "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework". In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, editors A. Yonezawa and B.C. Smith, pages 36-47, 1992.

[5] Takuo Watanabe and Akinori Yonezawa. "Reflection in an Object-Oriented Concurrent Language". In *Proceedings of OOPSLA '88*, pages 306-315, September 1988.

[6] Yuuji Ichisugi, Satoshi Matsuoka and Akinori Yonezawa. "RbCl: A reflective object-oriented concurrent language without a run-time kernel". In *Proceedings of the International Workshop on Reflection and Meta-level Architecture,* pages 24-35, November 1992. Tokyo, Japan.

[7] Bert Robben, Frank Piessens, Wouter Joosen. "Formalizing Correlate: from Practice to Pi". In *Proceedings of to the second BCS-FACS Northern Formal Methods Workshop*, july 1997, Ilkley, UK. Published electronically, http://ewic.springer.co.uk/.

[8] Jeff McAffer. "Meta-Level Programming with CodA". In *Proceedings of ECOOP'95*, pages 190-214, Aarhus, August 1995.

[9] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In *ACM Transactions on Computer Systems*, vol 1, nr 1, pages 63-75. February 1985.

[10] Philippe Mulet, Jacques Malenfant, Pierre Cointe. "Towards a Methodology for Explicit Composition of MetaObjects". In *Proceedings of OOPSLA'95*, pages 316-330, Austin, 1995.

[11] Smith, B.C. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pp 23-35, January 1984.

[12] G. Kiczales, J. des Rivieres and D. Bobrow. *The Art of the Meta-Object Protocol*, MIT Press, 1991.