OO-Motivated Process Algebra: A Calculus for CORBA-like Systems

Malcolm Tyrrell [*]
Andrew Butterfield
Alexis Donnelly
Dept. of Computer Science, Trinity College Dublin

October 20, 1999

Abstract

This paper is a proposal for a new two-tier calculus, designed to model aspects of CORBA-like systems at the CORBA object level. The higher object level known as Oompa abstracts away from the details of distribution (e.g. location and mobility), whereas the lower level, known as Loompa provides primitives for specifying these details. We present the syntax, and operational semantics, as well as two examples, a local invocation, and a remote one, to illustrate the features of the calculi. In the paper we also seek to justify our design decisions, both as to the nature of the calculi, and the need to develop same. A key strength of our approach is to maintain a close match between the level of object abstraction in Oompa/Loompa, and that found in conventional OO programming languages for CORBA systems.

1 Introduction

This paper is a proposal for a new two-tier calculus, designed to model aspects of CORBA-like systems at the CORBA object level. The higher object level known as Oompa abstracts away from the details of distribution (e.g. location and mobility), whereas the lower level, known as Loompa provides primitives for specifying these details. We present the syntax, and operational semantics, as well as two examples, a local invocation, and a remote one, to illustrate the features of the calculi. In the paper we also seek to justify our design decisions, both as to the nature of the calculi, and the need to develop same. A key strength of our approach is to maintain a close match between the level of object abstraction in Oompa/Loompa, and that found in conventional OO programming languages for CORBA systems. The work is being undertaken as part of the Formal Aspects of CORBA Systems (FACS) project — sponsored by Enterprise Ireland, Grant No. SC/97/631.

The structure of the rest of the paper is as follows: In §2 we introduce the problem domain; In §3 we look at other work done in this field; In §4 we discuss

[*]Supported by Enterprise Ireland Basic Research Grant No. SC/97/631

and motivate our choice of which aspects of CORBA we wish to consider; while in §5 we introduce the syntax and semantics of our calculus and extend it in §7.

We then present some examples in §6,8 discuss future work in §9 and present a summary and our conclusions in §10.

2 The Problem

The Object Management Group (OMG) developed the Common Object Request Broker Architecture (CORBA) [OMG98] to address various drawbacks in the then existing distributed system technology: too low level; too language or platform specific; and the presence of *legacy applications*. These issues prompted industry leaders to consider next generation communication technologies that would be versatile enough for large scale enterprise development. These technologies are now in widespread use and are being continually extended. Considering how complex they are, there is a vital need to understand them better so as to identify problems, to suggest improvements and to test the consistency of new extensions.

We will need sophisticated technological support to make them work, and we will need theory with which to understand them. The technological support will take the form of services which isolate the developers and users from the complexity. The theoretical support will provide analysis of systems, reliability and bring unseen issues to light. CORBA provides some of the required technology, but little in the way of theory. The aim of this work is to examine aspects of CORBA using a formal approach.

Formalisms and modeling languages bring some aspects of the system they are modeling to the fore where they can be examined and discussed. Other aspects are "abstracted away". In order to decide what type of formalism to use to model CORBA systems, we have to identify the features of the system that we need to bring out.

Our objective is to be able to model CORBA systems at the CORBA-object level, which means that programming and lower-level issues are less of a concern. Our aim is to match the level of abstraction in the calculus described here to that used in the CORBA object model.

In this report, we put forward a formal technique that can be used to analyse CORBA systems at this object-level. Here are some examples of the kind of issues we propose to model using our technique, were by "services" we mean both CORBAservices and CORBAfacilities

- Can a service offer all of its claimed functionality in the presence of other services?" This is an instance of *feature interaction*.
- Can a specific service be implemented independently of the ORB?
- Can services be re-used effectively in the implementation of other services?

3 Related Work

Our work is similar to that of others [Wal95], [Fei99] using the π-calculus to model various aspects of concurrent object-oriented systems. These models and ours all exploit π-calculus' seminal innovation of link-passing to model dynamic topologies. Unlike the *POOL* model, [Wal95] we do not attempt to model an entire concurrent object-oriented language. Our work differs from that on the COM model [Fei99] in opting to model (or ignore at will) aspects of distribution relevant for aspects of CORBA systems. In the latter work distribution aspects are not relevant, though it is shown (as in our example) how client and service objects may bind and interact. We believe our approach to dealing with or ignoring aspects of distribution is unique and practical.

There are standard ways that object-like behaviour can be coded up in the π-calculus using distinct channel names as references and setting up communication channels to send parameters through during an invocation. Notice, however, that to model a method invocation there are several communications. The interactional behaviour of a CORBA system would lead to an overwhelmingly complex model using such a scheme.

A π-calculus semantics has been given for the concurrent configuration language DARWIN [Eis93], which makes use of parallel composition and link passing to effect binding. Agents representing service requester and provider are composed with a special agent which transfers the name of one to the other. In our model, agents representing invoking and invoked objects interact directly in the local case. In the distributed case a proxy object is created on the invoking side which sends an agent to the remote side to effect the invocation and then returns to the original site. Using the location-based primitives of Loompa$_{\mathrm{X}}$ we thus model a typical CORBA interaction.

The location-based primitives of Loompa$_{\mathrm{X}}$ are inspired by those of Hennessey's D-π [HR98] where the focus of work is type-safe distributed computation. We use them to model distributed computations in CORBA systems, but as demonstrated in the D-π work, we also hope to exploit typing information, a topic for another paper. The *SEAL calculus* [Vit98], also based on the π-calculus, features agent mobility and strong protection mechanisms. Its focus is also secure distributed computations and the calculus features communication primitives that may specify channels located in another seal. Restrictions on the communications which may take place are motivated by security considerations. We take a more liberal approach since our concern is the modelling of CORBA systems.

Another design notation, based on π-calculus, $\pi o \beta \lambda$ [Jon93], focuses on the development of concurrent, object-oriented programs and uses formal techniques to reason about interference effects. The focus of our work has been to build simple models of aspects of CORBA systems using the same calculus.

The Distributed Join Calculus [FGL$^+$96] also extends π-calculus with explicit locations and primitives for agent mobility. The motivation appears to be the construction of models of representative mobile agent systems.

The Calculus of Actors [Agh86] is a higher-level formalism than the π-calculus although they have similar power. Actors allow much closer modeling of objects than π-calculus processes. An actor have a notion akin to object references, and have state. Unfortunately the state is, in a certain sense, bound to the "executing code". This is unsuited to the kind of client server behaviour that might exist in a CORBA system. Multiple clients may make calls upon one object, and it is not realistic to force the server to deal with them sequentially.

The *ambient calculus* [Car98] is a calculus that uses code mobility rather than communication as its method of interaction. It has a strongly defined notion of locality. However, the kind of location in the ambient calculus is really an abstract notion of an administrative domain rather than the kind we would want. It's hard to see how the ambient calculus could be put to use on our type of problems. It might be, however, very useful for modeling a mobility service at a high-level.

Another calculus that to consider is the *object calculus* [MA96]. This is a calculus that best models single threaded local object-based object-oriented systems. The single threading rules this out as a formalism for modeling CORBA systems. It nevertheless talks very usefully about typing issues associated with object systems.

4 Our View of CORBA

The definition of CORBA as found in the CORBA 2.2 standard [OMG98] is very complex, as it defines an industrial-strength software development platform. In order to make formal modelling feasible, we have chosen to simplify CORBA considerably, in order to focus most attention on the areas of interest. In this section we shall discuss briefly which aspects of CORBA have been retained, and which have been dropped — with some justifying remarks.

The first key simplification adopted is to stop considering multiple language systems. We introduce a single implementation language, Oompa, which was designed very to be simple and to be close to the CORBA notion of object. It is described in more detail §5.

In our simplified CORBA specification we use a reduced version of IDL, and provide a language mapping from this IDL to Oompa interfaces. We maintain IDL in some form as it is vital to so much of CORBA's behaviour. However, we reduce the number and type of datatypes that are available.

We also drop exceptions and context clauses from the notion of operation. The main justification is to reduce the complexity of systems. When we really need to model exceptions we can use an added return parameter with a standard set of values or wrap return values in some form of "Maybe" constructor.

However, we stress certain key features (albeit in a simplified form):

- Local Concurrency
  The CORBA specifications imply parallel behaviour when they discuss "server activation" policies, and many implementations use languages with threading, so we see concurrency at the pure object level. We say *local* concurrency to distinguish it from the inevitable concurrency that also emerges once we discuss distribution issues.

- Distribution
  A key function of CORBA systems is to provide a mechanism to allow

communication between different locations. This provision of this mechanism is of prime concern to the modelling of CORBA systems, and is an area we wish to discuss.

- State
  In modeling a CORBA system we will certainly need to discuss state. A CORBA system is more than just a sequence of object interactions, rather it performs state changes. While there are many ways of encoding state in various formal calculi that do not possess the explicit concept of a store, such encodings cause a huge increase in the model complexity. We view state as being an explicit part of the model where states and access to sates can be clearly and readily identified.

- Method Invocations
  We choose to model invocation directly rather than encode it up using another scheme. Similar to the case with state, we avoid complexity and gain the ability to easily identify method invocations.

- Classes
  We also choose to include classes. The impact of this is on the type system of our formalism, and the complexity brought in by having class types and subtypes can be dealt with statically before we consider the dynamic behaviour of our system. Classes allow us to follow a standard design style and keep close to the CORBA object model.

5 The Oompa Calculus

Oompa stands for OO-Motivated Process Algebra. It is an adaptation of the π-calculus which

- is typed,
- polyadic — so tuples can be sent and received,
- is object oriented.

This object orientation provides:

- classes that describe the attributes and methods of objects,
- a method invocation scheme,
- object state.
- interfaces, that list the method signatures.

An Oompa program (or *agent expression*), g, is executed (or *reduced*) in an environment consisting of: $\mathcal{C}$, the set of class definitions, which remains constant, and Δ, the current object state and typing information, which can change during the execution of an object. The current state of the system is the tuple consisting of g and Δ, which we denote $g\,\{\Delta$. Thus reduction statements take the form:

$$\mathcal{C} \mapsto g\,\{\Delta \longrightarrow g'\,\{\Delta'$$

The somewhat unusual "{" notation is handy when expressions become large, and "$\mapsto$" indicates that the transition depends on the set of class definitions.

An agent expression is a set of agents running in parallel. This is written

$$g_1 \mid g_2 \mid \dots g_n$$

An *agent* is executing code which is marked with the object from which it came. Agents have the form:

$$o[p]$$

The details of how the store, Δ, is managed is beyond the scope of this paper.

5.1 Code

There is a clear distinction in Oompa between the code that resides in class methods, and the behaviour of agents that it gives rise to. This contrasts with the π-calculus where all the code in the system already exists in some agent. The syntax reflects this: The `fork` primitive is an instruction to fork, rather that a description of two parallel agents. The syntax is also ASCII based, as Oompa is intended to be used as a simple programming language.

We give the set of primitive operations. The first five primitives are similar to the usual π-calculus primitives for null behaviour, parallel behaviour, channel creation, and send and receive on a channel. The remainder are object-based primitives unique to Oompa. The complete behaviour of these primitives is complicated, but their use is intuitive. Notice that the invocation and attribute access and update primitives ape the syntax of the send and receive primitives. This is intended to keep a consistent sense of directionality.

There are various other primitives we could have added to our Oompa system. Good examples are primitives for nondeterministic choice, replication, or a conditional structure. We choose to introduce Oompa in its simplest form, so these have been left out of the discussion.

5.1.1 Syntax for code

p	$::=$	$\texttt{nop}$	do nothing
		$\texttt{fork}\{p_1 \mid p_2\}$	fork
		$\texttt{new}\ c{:}T\ p_1$	create a new channel
		$c!\langle v_1, \dots v_n\rangle\ p_1$	send
		$c?(r_1{:}T_1, \dots r_n{:}T_n)\ p_1$	receive
		$o.m!\langle v_1, \dots v_n\rangle?(r_1, \dots r_m)\ p_1$	invoke a method
		$\texttt{create}\ o{:}T\ p_1$	create an object
		$a?(r)\ p_1$	access an attribute
		$a!\langle v\rangle\ p_1$	update an attribute

We will use the following notational conventions to simplify code expressions:

- We will write code of the form q `nop` as q.
- We will write $\texttt{fork}\{p_1|p_2|\dots|p_n\}$ for $\texttt{fork}\{p_1|\texttt{fork}\{p_2|\texttt{fork}\{\dots|p_n\}\}\}$.

5.2 Types

The details of the type system are beyond the scope of this paper. The basic syntax for types is given here:

5.2.1 Syntax for types

T	$::=$	P	primitive types
		$\text{chan}\langle T_1, \ldots T_n\rangle$	channel type
		I	interface type
		C	class type

5.3 Interfaces and Classes

Classes in Oompa have several roles. They contain method code, define attributes and their initial values, and describe class types.

Method definitions consist of a signature followed by their code. The signature defines the format a call of this method must take. Method code can contain two special syntactic entities, `this` and `return`. At method invocation time, when the code is copied into an agent, `this` will be bound to the containing object and `return` will be replaced by a newly created channel. This channel is used to pass the return values to the calling agent.

Initial values for attributes are defined in the class. This gives rise to a class initialisation function, from attribute names to initial values. For class C, this function is denoted C_f. Attributes are all private in Oompa.

5.3.1 Syntax for interfaces and classes

s	$::=$	$m?(r_1{:}T_1, \ldots r_n{:}T_n)$ $!\langle d_1{:}T'_1, \ldots d_m{:}T'_m\rangle$	signature
Id	$::=$	`interface` I $\{s_1 \ldots s_n\}$	interface definition
md	$::=$	$s\{p\}$	method definition
ad	$::=$	$a{:}T = \langle \mathit{literal}\rangle$	attribute declaration
Cd	$::=$	class C $\{ad_1 \ldots ad_m md_1 \ldots md_m\}$	class definition

5.4 Agents

Agents are the active components of an Oompa system. Code only describes behaviour; agents perform it. Agents are labelled with the object to which they belong, i.e. the object whose method they are running. This label permits access to the object's attributes to those agents belonging to that object.

Agents are created through forking and invocation.

5.4.1 Syntax for agents

$$
\begin{array}{rcll}
g & ::= & \texttt{nil} & \text{no behaviour} \\
 & & o[p] & \text{executing code } p \text{ from object } o \\
 & & (g_1 \mid g_2) & \text{parallel execution}
\end{array}
$$

5.5 Structural equivalence

An agent expression does not depend on the order or composition of the agents that make it up. Also, there is a `nil` agent which has no behaviour. As it has no effect, and systems can contain them with no discernible effect. The following structural equivalence rules summarise these points:

$$
\begin{array}{rcl}
(g_1 \mid g_2) & \equiv & (g_2 \mid g_1) \\
((g_1 \mid g_2) \mid g_3) & \equiv & (g_1 \mid (g_2 \mid g_3)) \\
o[\texttt{nop}] & \equiv & \texttt{nil} \\
(g_1 \mid \texttt{nil}) & \equiv & g_1
\end{array}
$$

5.6 Reduction Semantics

Two of the rules involve the creation of an agent. 5.6.4, where an agent splits in two, and 5.6.7, where a new agent is created to perform the method.

Most of these rules also have typing requirements, but we do not deal with the type system in this paper.

5.6.1 Left Equivalence

$$
\frac{g_1 \equiv g_2 \qquad \mathcal{C} \mapsto g_1 \{\Delta \longrightarrow g_1' \{\Delta'}{\mathcal{C} \mapsto g_2 \{\Delta \longrightarrow g_1' \{\Delta'}
$$

5.6.2 Right Equivalence

$$
\frac{g_1 \equiv g_2 \qquad \mathcal{C} \mapsto g_1' \{\Delta' \longrightarrow g_1 \{\Delta}{\mathcal{C} \mapsto g_1' \{\Delta' \longrightarrow g_2 \{\Delta}
$$

5.6.3 Parallel

$$
\frac{\mathcal{C} \mapsto g_1 \{\Delta \longrightarrow g_1' \{\Delta'}{\mathcal{C} \mapsto g_1 \mid g_2 \{\Delta \longrightarrow g_1' \mid g_2 \{\Delta'}
$$

5.6.4 Fork

When the fork instruction is executed, the agent becomes two, both of which belong to the same object as the original agent.

$$\mathcal{C} \mapsto o[\texttt{fork}\{p_1|p_2\}] \, \{\Delta \longrightarrow o[p_1] \mid o[p_2] \, \{\Delta$$

5.6.5 New Channel

To avoid some of the scoping issues of the π-calculus, the system chooses an completely unused name during channel and object creation. The channel name is added to the store along with its type.

$$\begin{array}{lll} \mathcal{C} \mapsto & o[\texttt{new}\ c\colon T\ p] & \{\Delta \\ & \longrightarrow & \\ & o[p\{\!|c'/c|\!\}] & \left\{ \begin{array}{l} \Delta \\ c'\colon T \end{array} \right. \end{array}$$

When c' is a new name.

5.6.6 Communication

Communication along a channel can occur between any two agents. This is essentially the same as the usual π-calculus rule.

$$\begin{array}{lll} \mathcal{C} \mapsto & \begin{array}{l} o_1[c!\langle v_1, \ldots v_n \rangle\ p_1] \mid \\ o_2[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2] \end{array} & \{\Delta \\ & \longrightarrow & \\ & \begin{array}{l} o_1[p_1] \mid \\ o_2[p_2\{\!|v_1/r_1, \ldots v_n/r_n|\!\}] \end{array} & \{\Delta \end{array}$$

5.6.7 Method Invocation

When an agent invokes a method, m, of object o_1, a new channel name, r, is chosen. The calling agent is blocked behind a receive on this channel. The code of the method, here p_1, is copied from $\mathcal{C}$ and the input parameters are substituted by their values. `this` is replaced by o_1, `return` is replaced by the new channel r.

$$\begin{array}{lll} \mathcal{C} \mapsto & o[o_1.m!\langle v_1, \ldots v_n \rangle ?(s_1, \ldots s_m)\ p] & \{\Delta \\ & \longrightarrow & \\ & \begin{array}{l} o[r?(s_1\colon T_1, \ldots s_m\colon T_m)\ p] \mid \\ o_1[p_1\{\!|v_1/r_1, \ldots v_n/r_n, o_1/\texttt{this}, r/\texttt{return}|\!\}] \end{array} & \left\{ \begin{array}{l} \Delta \\ r\colon \mathrm{chan}\langle T_1, \ldots T_m \rangle \end{array} \right. \end{array}$$

When r is a new name.

5.6.8 Object Creation

Object creation involves the choosing a new name for the object. The new object name is added to the store with its class and its assignment function is set to the class initialisation function.

$$\begin{array}{lll} \mathcal{C} \mapsto & o[\texttt{create}\ o_1{:}C\ p] & \{\Delta \\ & \longrightarrow & \\ & o[p\{\!|o_1'/o_1|\!\}] & \left\{\begin{array}{l}\Delta \\ o_1'{:}C = C_f\end{array}\right. \end{array}$$

When o_1' is a new name.

5.6.9 Attribute Access

This replaces occurrences of r in code p by the attribute value. Agents can only access the state of the object to which they belong.

$$\mathcal{C} \mapsto o[a?(r)\ p]\,\{\Delta \longrightarrow o[p\{\!|v/r|\!\}]\,\{\Delta$$

When $\Delta.o(a) = v$.

5.6.10 Attribute Update

This simply updates the attribute assignment function of the object. Agents can only update the state of the object to which they belong.

$$\mathcal{C} \mapsto o[a!\langle v\rangle\ p]\,\{\Delta \longrightarrow o[p]\,\{\Delta'$$

When $\Delta'.o_1(a_1) = \left\{\begin{array}{ll} \Delta.o_1(a_1) & \text{if } o_1 \neq o, a_1 \neq a \\ v & \text{otherwise}\end{array}\right.$

6 Examples

6.1 An example — Updating a Cell

We present a simple example of Oompa — a simple cell and a client class that makes use of it. The Cell class is coded as follows:

```
class Cell
{
  Integer store = 0

  get_contents?()!<value:Integer>
  {
    store?(current_value)
    return!<current_value>
  }

  set_contents?(value:Integer)!<>
  {
    store!<value>
    return!<>
  }
}
```

It has a single attribute, `store`, which holds an integer. Attributes are private in Oompa, so this value is only accessible through methods, in this case `get_contents` and `set_contents`. The following code describes a simple client:

```
class Example
{
  main?()!<>
  {
    create a:Cell
    a.set_contents!<5>?()
    this.check_value!<a>?<v>
    return!<>
  }

  check_value?(theCell:Cell)!<val:Integer>
  {
    theCell.get_contents!<>?(curValue)
    return!<curValue>
  }
}
```

The example behaves as follows: The main method is invoked, and it creates a new Cell object, *a*. It invokes the `set_contents` method of the cell to change the value its store to 5. It then invokes the class's own `check_value` method, passing as the name of the Cell object as a parameter. This method requests the value of the cells contents, and then returns. The main method then finishes.

We give the formal behaviour of the system as a series of Oompa reduction steps. The state of the system is described by the set of classes, the set of agents and the contents of the store, which holds typing information and attribute assignments. Notationally, we omit the set of classes as this is constant.

The system starts by creating an object of the `Example` class, called `root`, and invoking its main method. By rule 5.6.7, this involves

- choosing a new channel name for return values, in this case *ret1*,
- copying the method code from the `Example` class,
- applying the required substitutions,
- placing the resulting code in an agent labelled with the object to which it belongs.

In this case the substitutions replace `return` by *ret1* and `this` by *root*.

The type information of the new channel is added to the data store. Also, the class of the object is added along with an assignment function which gives values to its attributes (in this case none).

$$\begin{array}{l} root[\\ \quad \texttt{create}\ a\colon Cell \\ \quad a.set\_contents!\langle 5\rangle?() \\ \quad root.check\_value!\langle a\rangle?(v) \\ \quad mainRet!\langle\rangle] \end{array} \quad \left\{ \begin{array}{l} mainRet\colon \mathrm{chan}\langle\rangle \\ root\colon Example = \emptyset \end{array} \right.$$

The only agent in the system begins with a create operation, so rule 5.6.8 must apply. This chooses a new name for the object, substituting that name for the occurrences in the agent body. The class of this object, along with its initial assignment function, is added to the data store.

$$\longrightarrow \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{a1}.\mathit{set\_contents}!\langle \mathit{5}\rangle?() \\ \quad \mathit{root}.\mathit{check\_value}!\langle \mathit{a1}\rangle?(v) \\ \quad \mathit{mainRet}!\langle\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 0] \end{array} \right.$$

Next is the invocation on the cell $a1$, so rule 5.6.7 applies. The return channel *ret1* is created and the method code is copied into an agent labelled with the object name, $a1$. The value 5 is substituted throughout the agent for the input parameter *value* and `return` is replaced by *ret1*. The calling agent is blocked by a receive on the *ret1* channel, waiting for the method to return. *ret1*'s typing is added to the store.

$$\longrightarrow \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{ret1}?() \\ \quad \mathit{root}.\mathit{check\_value}!\langle \mathit{a1}\rangle?(v) \\ \quad \mathit{mainRet}!\langle\rangle] \mid \\ \mathit{a1}[\\ \quad \mathit{store}!\langle 5\rangle \\ \quad \mathit{ret1}!\langle\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 0] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \end{array} \right.$$

The main agent is blocked, so the only available action is the attribute update performed by the Cell $a1$, using rule 5.6.10. The assignment function for the object $a1$ now assigns 5 to the *store* attribute.

$$\longrightarrow \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{ret1}?() \\ \quad \mathit{root}.\mathit{check\_value}!\langle \mathit{a1}\rangle?(v) \\ \quad \mathit{mainRet}!\langle\rangle] \mid \\ \mathit{a1}[\\ \quad \mathit{ret1}!\langle\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 5] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \end{array} \right.$$

The two agents in the system synchronise on the *ret1* channel, using rule 5.6.6. This represents the finishing of the `set_contents` method of $a1$. This method returns no values, so no data is communicated. After this, the agent *a1*[`nop`] is tidied away using two applications of equivalence, rule 5.6.2.

$$\longrightarrow^* \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{root}.\mathit{check\_value}!\langle \mathit{a1}\rangle?(v) \\ \quad \mathit{mainRet}!\langle\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 5] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \end{array} \right.$$

Next, the invocation of the *root* object's `check_value` method occurs, using rule 5.6.7. This involves creating a new return channel, substituting occurrences of the input parameter `theCell` by *a1* in the cell and replacing `return` by *ret2*.

$$\longrightarrow \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{ret2}?(v\colon \mathit{Interger}) \\ \quad \mathit{mainRet}!\langle\rangle] \mid \\ \mathit{root}[\\ \quad \mathit{a1}.\mathit{get\_contents}!\langle\rangle?(\mathit{curValue}) \\ \quad \mathit{ret2}!\langle \mathit{curValue}\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 5] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \\ \mathit{ret2}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \end{array} \right.$$

The calling code is blocked by a receive on *ret2*, so the next transition that occurs is the invocation on *a1*.

$$\longrightarrow \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{ret2}?(v\colon \mathit{Interger}) \\ \quad \mathit{mainRet}!\langle\rangle] \mid \\ \mathit{root}[\\ \quad \mathit{ret3}?(\mathit{curValue}\colon \mathit{Integer}) \\ \quad \mathit{ret2}!\langle \mathit{curValue}\rangle] \mid \\ \mathit{a1}[\\ \quad \mathit{store}?(\mathit{current\_value}) \\ \quad \mathit{ret}3!\langle \mathit{current\_value}\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 5] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \\ \mathit{ret2}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \\ \mathit{ret3}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \end{array} \right.$$

The rule that applies here is 5.6.9, which substitutes the occurrences of the given name for the current value of that attribute. In this case the one occurrence of *current_value* is replaced by 5.

$$\longrightarrow \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{ret2}?(v\colon \mathit{Interger}) \\ \quad \mathit{mainRet}!\langle\rangle] \mid \\ \mathit{root}[\\ \quad \mathit{ret3}?(\mathit{curValue}\colon \mathit{Integer}) \\ \quad \mathit{ret2}!\langle \mathit{curValue}\rangle] \mid \\ \mathit{a1}[\\ \quad \mathit{ret}3!\langle 5\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 5] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \\ \mathit{ret2}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \\ \mathit{ret3}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \end{array} \right.$$

The 5.6.6 rule applies here, as the `get_contents` method of the cell finishes up and returns its value. Two applications of 5.6.2 are also used to tidy away the empty agent.

$$\longrightarrow^* \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{ret2}?(v\colon \mathit{Interger}) \\ \quad \mathit{mainRet}!\langle\rangle] \mid \\ \mathit{root}[\\ \quad \mathit{ret2}!\langle 5\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 5] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \\ \mathit{ret2}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \\ \mathit{ret3}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \end{array} \right.$$

This is similar to the previous step.

$$\longrightarrow^* \quad \begin{array}{l} \mathit{root}[\\ \quad \mathit{mainRet}!\langle\rangle] \end{array} \quad \left\{ \begin{array}{l} \mathit{mainRet}\colon \mathrm{chan}\langle\rangle \\ \mathit{root}\colon \mathit{Example} = \emptyset \\ \mathit{a1}\colon \mathit{Cell} = [\mathit{store} \mapsto 5] \\ \mathit{ret1}\colon \mathrm{chan}\langle\rangle \\ \mathit{ret2}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \\ \mathit{ret3}\colon \mathrm{chan}\langle \mathit{Integer}\rangle \end{array} \right.$$

As no mechanism has been described to pick up the return value of the main method, no rule applies, and we deem that the program has finished.

7 Loompa

Loompa stands for “located Oompa” and is an extension of Oompa which incorporates the idea of locality. An objects in Loompa reside at a specific location, and never leaves. Agents can only run at the location where their object reside.

Loompa is actually a base calculus which provides no more interesting behaviour than Oompa, as agents at different locations have no way of interacting. The intention is to extend Loompa with different remote interaction primitives to suit the modeling problem being dealt with. We will mention Loompa$_\mathrm{X}$ as an extension which provides a form of code mobility.

Loompa systems are described by a number of parallel location expressions:

$$l_1 \mid \ldots l_n$$

These location expressions take the form

$$loc^{\mathcal{C}}[\![g]\!]$$

Note that each location is marked with its set of classes. The set of classes at different locations need not be the same.

The object state and typing information of the system is managed as a dictionary recording the state at each location.

$$\begin{cases} {loc_1}^{\mathcal{C}_1} = \{\Delta_1 \\ \vdots \\ {loc_n}^{\mathcal{C}_n} = \{\Delta_n \end{cases}$$

We usually denote this dictionary Γ. The dynamic part of the system is expressed as a tuple

$$l\,\{\Gamma$$

so a Loompa reduction is written:

$$l\,\{\Gamma \longrightarrow l'\,\{\Gamma'$$

7.1 Semantics

These is the usual structural equivalence rules:

$$\begin{array}{rcl} (l_1 \mid l_2) & \equiv & (l_2 \mid l_1) \\ ((l_1 \mid l_2) \mid l_3) & \equiv & (l_1 \mid (l_2 \mid l_3)) \\ loc^{\mathcal{C}}[\![\mathtt{nil}]\!] & \equiv & NIL \\ (l_1 \mid NIL) & \equiv & l_1 \\ loc^{\mathcal{C}}[\![g_1 \mid g_2]\!] & \equiv & loc^{\mathcal{C}}[\![g_1]\!] \mid loc^{\mathcal{C}}[\![g_2]\!] \end{array}$$

and the usual reduction rules of left/right equivalence and parallel behaviour.

$$\frac{l_1 \equiv l_2 \quad l_1 \{\Gamma \longrightarrow l_1' \{\Gamma'}{l_2 \{\Gamma \longrightarrow l_1' \{\Gamma'}$$

$$\frac{l_1 \equiv l_2 \quad l_1' \{\Gamma \longrightarrow l_1 \{\Gamma'}{l_1' \{\Gamma \longrightarrow l_2 \{\Gamma'}$$

$$\frac{l_1 \{\Gamma \longrightarrow l_1' \{\Gamma'}{l_1 \mid l_2 \{\Gamma \longrightarrow l_1' \mid l_2 \{\Gamma'}$$

7.2 Local Behaviour

Loompa's key behaviour is "local behaviour". Essentially, each location can carry out its business as if it were an Oompa system. The basic form of the local behaviour rule is:

$$\frac{\mathcal{C} \mapsto g \{\Delta \longrightarrow g' \{\Delta'}{\mathit{loc}^{\mathcal{C}} [\![g]\!] \Big\{ \mathit{loc}^{\mathcal{C}} = \{\Delta \quad \longrightarrow \quad \mathit{loc}^{\mathcal{C}} [\![g']\!] \Big\{ \mathit{loc}^{\mathcal{C}} = \{\Delta'}$$

However, there are side conditions for some of the Oompa reductions. The main differences are

- Creation of new channels and new objects: To guarantees uniqueness of names, the chosen named are marked with the location.
- Channel communication can only occur at the location where the channel was created.
- Method invocation replaces the keyword `here` with the current location.

7.3 Loompa$_{\mathrm{X}}$

We consider one main extension of Loompa here, Loompa$_{\mathrm{X}}$. This adds a simple code migration primitive called `execute@`. This takes a location and a piece of code and starts the code running on the location in an agent labelled *guest*.

$$\mathit{loc_1}^{\mathcal{C}_1} [\![o[\texttt{execute@}\ \mathit{loc_2}\{p\}]]\!] \{\Gamma \quad \longrightarrow \quad \mathit{loc_2}^{\mathcal{C}_2} [\![\mathit{guest}[p]]\!] \{\Gamma$$

8 Distribution Example

We now demonstrate how Loompa can be used to exhibit distributed system behaviour by giving an example of a mechanism for remote method invocation in Loompa$_{\mathrm{X}}$.

We consider two locations *loc1* and *loc2*. A server at *loc2* has created a Cell object named *myImpl1*. The implementor has published the Cell's interface and made the name and location of the Cell object available. The client will use this interface to make invocations on the remote object.

```
interface Cell_Interface
{
  get_contents?()!<value:Integer>
  set_contents?(value:Integer)!<>
}
```

We give an example client here:

```
class Client
{
  main?()!<>
  {
    create a:Cell_Proxy
    a.bind!<loc2,myImpl1>?()
    this.check_value!<a>?(v)
    return!<>
  }

  check_value?(theCell:Cell_Interface)!<val:Integer>
  {
    theCell.get_contents!<>?(curValue)
    return!<curValue>
  }
}
```

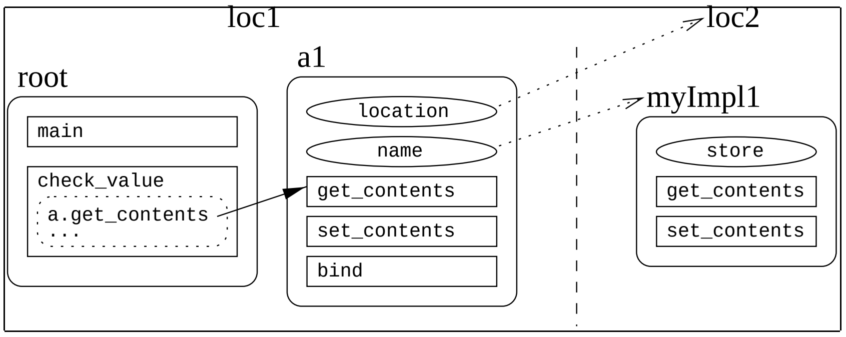
Notice the similarity between this and the Oompa example in 6.1. Instead of creating a real Cell object, the main code creates a proxy for the remote class. This proxy is bound to the remote Cell implementation with the `bind` call. The `check_value` method constitutes the client proper. The only change between this and the local case is the use of the interface `Cell_Interface` in place of the class `Cell`. To the client, the remote invocation appears essentially the same as the local case.

Also notice that the client is syntactically an Oompa class. Even though it will run in a distributed Loompa system, the client implementor is isolated from the details of distribution.

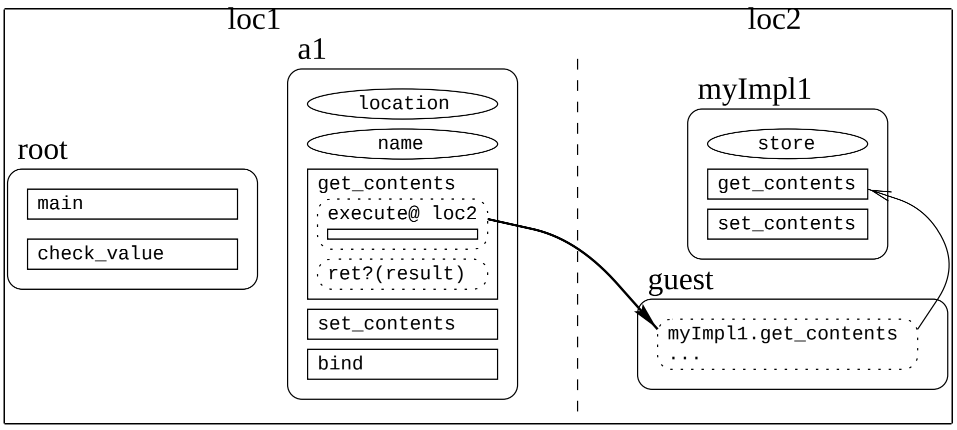
We give some details of the proxy class, illustrating how the `execute@` primitive of Loompa$_X$ provides us with sufficient power to perform the remote invocation. It is expected that the proxy class would be generated automatically from the interface, in a system where remote method invocation is provided.

```
class Cell_Proxy
{
  Location location = null
  Cell_Interface name = null

  get_contents?()!<value:Integer>
  {
    new result_chan:Integer
    fork
    {
      location?(dest)
      name?(target)
      execute@ dest
      {
        target.get_contents!<>?(val)
        execute@ here
        {
          result_chan!<val>
        }
      }
    |
      result_chan?(result)
      return!<result>
    }
  }

  set_contents?(value:Integer)!<>
    ...similar to get_contents.

  bind?(dest:Location,target:Object)!<>
  {
    location!<dest>
    name!<target>
    return!<>
  }
}
```

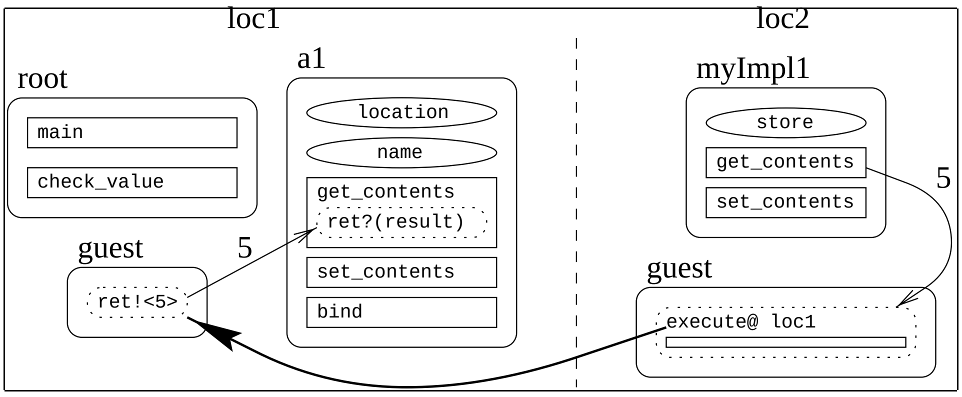
We now show, diagrammatically, how the system behaves. Assume that the client has been instantiated as an object called *root*. It creates the proxy object, *a1*, and binds it to the remote object. In the diagram we see that the *location* and *name* attributes refer to the location and name of the remote object. It then invokes its own `check_value` method, which makes an invocation on its `Cell_Interface` reference. This is actually an invocation on the proxy method `get_contents` as we see in the diagram.



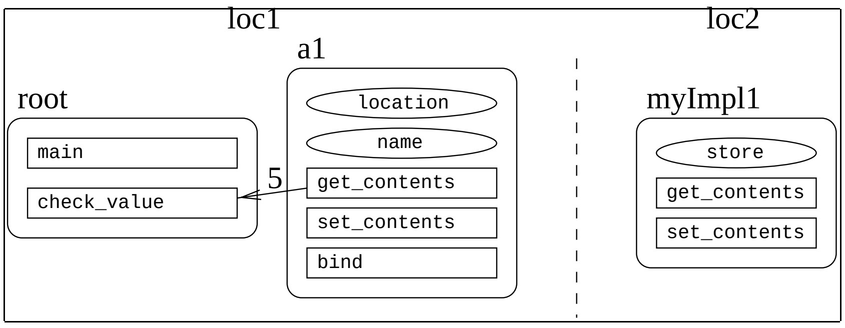
This proxy method `get_contents` creates a channel, called *ret*, and forks. One of the agents waits on the *ret* channel, while the other performs an `execute@` operation which causes creates a guest agent at *loc2*. This agent invokes the real Cell's `get_contents` method.



The Cell returns the value 5 to the guest agent which performs an `execute@` operation to return it to *loc1*. This agent sends the value 5 to the agent waiting on channel *ret*.



This agent then performs the `return` of the proxies `get_contents` method, which returns the value 5 to the client.



To the client, unaware of the distributed behaviour, this seems like a local communication.

9 Future Work

We are currently exploring the modelling of further object scenarios using Oompa/ Loompa to provide further feedback on the practicality of our approach. Some of these will involve exploiting the location aspects of Loompa in client-server settings of frequent practical interest. Another fruitful area might be simple distributed algorithms which lend themselves to an object-oriented implementations in which location must be modelled.

We hope to explore the use of typing information associated with objects in the fashion of Hennessey's D-π [HR98]. The focus of our interest however, is in providing additional support for (type-safe) component-wise construction of software systems. CORBA currently provides little semantic support in this area through IDL. The thrust of the D-π work has been to provide run-time security through type checking. It may be possible to exploit or adapt existing work on typed extensions to the π-calculus for our purposes [PS96], [Wal95], [PT98]. The use of the subtype relation to capture notions of substitutability may be useful in structuring systems based on components [PS96].

10 Conclusions

In this paper we present and motivate our approach to modelling relevant aspects of CORBA systems based on the π-calculus. We demonstrate how the Oompa notation can be used as a shorthand notation to bridge the "impedance gap" between a programming language level notation and that of the π-calculus. We illustrate the transition rules for our notation, based essentially on the communication primitives of π-calculus. We deal with the physical aspects of distribution using the `execute@` primitive of the related Loompa notation. Although it is still at an early stage, we believe our approach may offer some benefits in potentially bringing "rigour without pain" to the practice of constructing distributed object-based systems.

References

[Agh86] Gul Agha. *Actors, a Model of Concurrent Computation in Distributed Systems.* MIT press, 1986.

[Car98] A Cardelli, L & Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures*, LNCF, pages 140–155. Springer Verlag, 1998.

[Eis93] R Eisenbach, S & Patterson. π-calculus semantics for concurrent configuration language darwin. In *Proceedings of the Hawaii International Conference on System Sciences*, 1993.

[Fei99] Loe Feijs. Modelling microsoft com using the π-calculus. In J. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods 99*, LNCS, pages 1343–1363. Springer-Verlag, 1999.

[FGL$^+$96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer.

[HR98] Matthew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. Technical report, University of Sussex, May 1998.

[Jon93] C. B. Jones. A π-calculus semantics for an object-based design notation. In E Best, editor, *Proceedings of CONCUR'93*, LNCS, pages 158–172. Springer-Verlag, 1993.

[MA96] Luca Cardelli Martin Abadi. *A Theory of Objects*. Springer-Verlag, 1996.

[OMG98] OMG. *CORBA 2.2 Specification*, February 1998.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, pages 409–453, October 1996.

[PT98] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. CSCI Technical Report 476, Indiana University, March 1998.

[Vit98] G Vitek, J & Castagna. Towards a calculus of secure mobile computations. In *Proceedings of IEEE Workshop on Internet Programming Languages 1998*. IEEE, 1998.

[Wal95] David Walker. Objects in the pi-calculus. *Information and Computation*, 1995.