A Formal Framework for Modelling and Analysing Mobile Systems

Graeme Smith

School of Information Technology and Electrical Engineering
University of Queensland, Australia
Email: smith@itee.uq.edu.au

Abstract

This paper presents a formal framework for modelling and analysing mobile systems. The framework comprises a collection of models of the dominant design paradigms which are readily extended to incorporate details of particular technologies, i.e., programming languages and their run-time support, and applications. The modelling language is Object-Z, an extension of the well-known Z specification language with explicit support for object-oriented concepts. Its support for object orientation makes Object-Z particularly suited to our task. The system structuring techniques offered by object orientation are well suited to modelling mobile systems. In addition, inheritance and polymorphism allow us to exploit commonalities in mobile systems by defining more complex models in terms of simpler ones.

Keywords: formal methods, mobile computing, object orientation

1 Introduction

Recent developments in both hardware and software technology have provided the basis for a new distributed computing paradigm — *mobile computing*. Portable hardware devices such as laptops and smart cards enable computer networks where physical devices migrate from one location to another. This ability is further enhanced by advances in wireless data communication and the embedding of computers in a wide range of devices such as active badges, pagers, mobile phones and GPS receivers. To overcome the latency and bandwidth restrictions of the Internet, programming languages such as Telescript [White, 1996] and Java Aglets [Lange, 1997] have been developed which allow executable code to similarly migrate throughout a network.

In such a setting, issues such as access to distributed data and services, routing, consistency of distributed data and, in particular, the secure transfer of confidential data and code arise and must be dealt with. To tackle these issues in the design of traditional distributed, i.e., non-mobile, computing applications is a complex task involving aspects of data, (network) architecture, behaviour, interaction and distribution. For mobile systems, another layer of complexity is introduced by the fact that the architecture and distribution aspects of the modelled application can change dynamically. If we are to move beyond trivial applications of the mobile computing paradigm, we need a sound understanding of the base concepts of the paradigm and a way of identifying and tackling the complex issues that arise in given applications.

Toward this end, Fuggetta, Picco and Vigna [Fuggetta et al., 1998] present a conceptual framework for understanding and guiding development of systems incorporating mobile code. Their framework introduces three dimensions along which such systems can be classified: *technologies*, *design paradigms* and *applications*. Mobile code technologies are the mechanisms present in languages to enable and support code mobility. Design paradigms are the architectural styles that enable the technologies to be effectively utilised. Applications are systems which use mobile code. In particular, four design paradigms are described: the *Client-Server* paradigm which supports non-mobile distributed computing; the *Remote Evaluation* and *Code on Demand* paradigms which support *weak mobility*, i.e., where code, but not execution state, can move around a network; and the *Mobile Agent* paradigm which supports *strong mobility*, i.e., where both code and execution state can move around a network.

While this informal framework has been shown to be useful for designing mobile systems [Baldi et al., 1997, Ghezzi and Vigna, 1997], it is not meant for reasoning about such systems. As the authors point out, "models enabling formal reasoning and verification" are also needed. In this paper, we take up this challenge by producing a formal framework for modelling mobile code technologies, design paradigms and applications using the specification language Object-Z [Smith, 2000].

Object-Z is an extension of the well-known Z specification language [Spivey, 1992] with explicit support for object-oriented concepts such as classes, inheritance, object references (which allow for aliasing and mutual recursion between objects) and polymorphism. These concepts facilitate our task in two ways.

1. The system structuring techniques offered by object orientation are well suited to modelling mobile systems.

We model network nodes as objects and connections between nodes via object references. Changing references hence changes network connectivity enabling dynamic network topologies, as would exist in a wireless network, to be modelled. Processes are also modelled as objects and are associated uniquely with a node via a notion of object containment. Such processes can be passed as parameters between nodes modelling process migration. Mutually recursive references between two objects allow either of the objects to activate an event, in which the other is a passive participant. This is useful to model, for example, whether a process moves autonomously or is sent to another node by its host node.

2. Inheritance and polymorphism allow us to exploit commonalities in mobile systems by defining

more complex models in terms of simpler ones.

We provide a library of design paradigm models which can be readily extended to model new paradigms as they arise. We also discuss how this library enables us to define particular technologies and applications by incorporating their details into our models.

We begin in Section 2, with a brief introduction to Object-Z, including conventions for interpreting specifications which we adopt in our framework. In Section 3, we specify a generic network and illustrate how the common Client-Server paradigm of distributed computing can be defined from it via inheritance. In Section 4, we extend the Client-Server model to capture the design paradigms supporting weak mobility, i.e., the Remote Evaluation and Code on Demand paradigms. In Section 5, we similarly extend the Client-Server model to capture the Mobile Agent paradigm supporting strong mobility. In Section 6, we discuss other approaches to modelling mobile systems. In Section 7, we conclude with a brief discussion of future directions.

2 Object-Z

Object-Z [Smith, 2000] is a formal specification language which supports modelling using object-oriented concepts. An Object-Z specification comprises a number of global constant and type definitions, specified according to the syntax of Z [Spivey, 1992], and a number of classes including a *system class* which acts as the interface to the system being modelled. A full description of Object-Z can be found elsewhere [Smith, 2000]. Below we summarise those aspects of the language relevant to this paper.

A class encapsulates a collection of typed state variables, invariant constraints on these variables, initial constraints on these variables and several operations which may change these variables. The type of a state variable may be a class. In this case, the value of the variable is a reference to an object of that class. Using references to objects, rather than values of objects, allows aliasing and mutually recursive class specifications.

To indicate that an object is (physically) contained by another class, rather than just referenced by it, its type is annotated with the symbol ©. For example, the declaration $a : A_{©}$ in class B indicates that an object of class B has a reference a to an object of class A and that this object is contained by the object of class B. A single object cannot be (directly) contained by more than one object.

Classes may be specified incrementally using inheritance. A class which inherits another may extend its definition with new state variables, new invariant and initial constraints, and new operations. It may also add new constraints to existing operations. In a specification including one or more inheritance hierachies, variables may be declared polymorphically to have the type of any class within a hierarchy.

As in Z, it is useful to adopt conventions for interpreting Object-Z specifications when used for a particular type of modelling. In our framework, we adopt two conventions for interpreting Object-Z specifications.

Firstly, we assume that the modelled system comprises those objects that can be reached (directly or indirectly) via object references explicitly declared in the system class. For example, if A is the system class of the specification below, then the modelled system comprises one object of class B and two objects of class C. (An alternative convention, which we do not adopt, would include an object of the system class A in the modelled system.)

A
- $b : B$
- $\dots$

B
- $c_1, c_2 : C$
- $\dots$

C
- $\dots$

Our second convention concerns which object activates an event which involves two or more objects. This is done by objects referencing (either directly or indirectly) all objects involved in the events which they activate. In the system class, operations are specified in terms of the active object only. The participation of other objects in an event are specified in the active object's class via the object references. For example, in the specification below, the system class A declares two objects b_1 and b_2 which reference each other. The operations $OpA1$ and $OpA2$ involve both objects, but whereas $OpA1$ is activated by b_1, $OpA2$ is activated by b_2.

A
- $b_1, b_2 : B$
- $b_1.b = b_2$
- $b_2.b = b_1$
- $OpA1 \mathrel{\widehat=} b_1.OpB2$
- $OpA2 \mathrel{\widehat=} b_2.OpB2$

B
- $b : B$
- $OpB1 \mathrel{\widehat=} \dots$
- $OpB2 \mathrel{\widehat=} b.OpB1$

Formal rules for reasoning about Object-Z specifications have been developed [Smith, 1995a], as have approaches to reasoning which take advantage of the object-oriented structure of specifications [Smith, 1995b, Griffiths, 1997]. Work on tools to assist reasoning in Object-Z are also being developed. These include tools for *animation* (interactive exploration of specifications) [McComb and Smith, 2003], *model checking* (automatic proof on specifications with a limited state space) [Kassel and Smith, 2001] and *theorem proving* (interactive proof on specifications, including those with a large, or even infinite, state space) [Smith et al., 2002]. There is also ongoing work looking at combining model checking and theorem proving techniques [Smith and Winter, 2003, Winter and Smith, 2003].

3 Distributed Systems

In this section, we specify a generic network in Object-Z and show how it can be extended, via inheritance, to model the common Client-Server paradigm.

We begin by declaring two types, *Address* and *Data*, which will be used throughout the paper.

$[Address, Data]$

We also define a type *Message* to denote any type of message that can be sent on a network. A message has three components: a source, a destination and some data. It may represent a single packet in a network or several packets whose data parts need to be combined at the destination.

$Message$
- $source, destination : Address$
- $data : Data$

3.1 A Generic Network

Given these types, we define the class of a generic network node at the top of Figure 1.

Node

$address : Address$
$input\_queue : \mathrm{seq}\, Message$
$neighbours : \mathbb{P}\, Node$
$next : Address \rightarrow \mathbb{P}\, Node$

$\mathrm{ran}\, next \subseteq neighbours$

INIT

$input\_queue = \langle\, \rangle$

Send$_0$

$m! : Message$

$m!.source = address$

Receive

$\Delta(input\_queue)$
$m? : Message$

$input\_queue' = input\_queue \frown \langle m? \rangle$

Accept

$\Delta(input\_queue)$

$input\_queue \neq \langle\, \rangle$
$(head\ input\_queue).destination = address$
$input\_queue' = tail\ input\_queue$

Transfer$_0$

$\Delta(input\_queue)$
$m! : Message$

$input\_queue \neq \langle\, \rangle$
$(head\ input\_queue).destination \neq address$
$input\_queue = \langle m! \rangle \frown input\_queue'$

$Send \mathrel{\widehat=} [\, nn : neighbours\,] \bullet (Send_0 \bullet [\, nn \in next(m!.destination)\,]) \parallel_! nn.Receive$
$Transfer \mathrel{\widehat=} [\, nn : neighbours\,] \bullet (Transfer_0 \bullet [\, nn \in next(m!.destination)\,]) \parallel_! nn.Receive$

Network

$nodes : \mathbb{P} \downarrow Node$

$\forall\, n : nodes \bullet n.neighbours \subseteq nodes$
$\forall\, n_1, n_2 : nodes \bullet n_2 \in n_1.neighbours \Rightarrow n_1 \in n_2.neighbours$

INIT

$\forall\, n : nodes \bullet n.\mathit{INIT}$

$Send \mathrel{\widehat=} [\, n : nodes\,] \bullet n.Send$
$Accept \mathrel{\widehat=} [\, n : nodes\,] \bullet n.Accept$
$Transfer \mathrel{\widehat=} [\, n : nodes\,] \bullet n.Transfer$

Figure 1: Generic network.

Each node has an address, an input queue of messages (used to model asynchronous communication), a set of neighbours and a routing strategy. These are represented by state variables. The routing strategy, represented by state variable *next*, is a function which, given a destination address, returns a set of nodes which are the possible next nodes a message should be sent to in order to reach the destination. The set of nodes can be empty; when the address is the current node's address, for example. An invariant constraint restricts the nodes in the set of nodes to be neighbouring nodes. ($\mathrm{ran}\, f$ denotes the range of function f.)

Initially, the input queue is empty and the values of the other variables are not further constrained. The class has six operations.

$Send_0$ outputs a message $m!$ (the ! decoration denotes an output variable) whose source is the node's address. The operation captures just the conditions common to nodes in all kinds of networks. For specific networks, more detail about the data which is sent, or the reason for sending a message may be required. In these cases, further constraints could be added to this, and other operations, using inheritance.

Receive inputs a message $m?$ (the ? decoration denotes an input variable) and adds it to the input queue. (The Δ symbol in an operation precedes a list of state variables which the operation may change. All other state variables are unchanged. The variables after an operation are decorated with a $'$.) When the input queue is not empty, the message at its head may be accepted by the node or transferred to another node depending on its destination. The former situation is captured by *Accept* and the latter by $Transfer_0$.

The final two operations *Send* and *Transfer* combine previously defined operations with the operators $\bullet$ ("such that") and $\|_!$ ("in parallel with"). They specify the sending and transferring of a message, respectively, to a neighbouring node *nn* which receives the message. (The parallel operator $\|_!$ equates inputs to outputs with common basenames, i.e, apart from the ! and ? decorations.)

The class of a generic network is defined at the bottom of Figure 1.

This class is the system class of our specification. Hence, our system comprises a set of nodes. The $\downarrow$ preceding the type *Node* is the polymorphism symbol. It indicates that the nodes may be of type *Node* or any class derived from *Node* via inheritance. This allows us to extend *Node* with additional variables and constraints, to define specific design paradigms, for instance, and still use the *Network* class. The invariant constraints ensure that the neighbours of all nodes in a network are also in the network, and that for any pair of nodes, if the second is a neighbour of the first then the first is a neighbour of the second.

Initially, all nodes satisfy the initial constraints of their class. Therefore, initially their input queues are empty. Three operations are possible in a generic network. *Send* models a node n proactively sending a message. This will involve another node passively receiving the message (as specified in the *Send* operation of *Node*). *Accept* models a node n proactively accepting a message in its input queue. *Transfer* models a node n proactively transferring a message (which is passively received by another node as specified in the *Transfer* operation of *Node*).

Although *Network* defines a statically connected network, it can be used as the starting point for defining a dynamically connected network. *Node* could be extended (via inheritance) to have an operation which allows the variables *neighbours* and *next* to change, and an operation to direct its neighbours to appropriately update their *neighbours* and *next* variables while it does this. The *Network* class would then be extended to allow a node n to proactively perform the second operation.

3.2 The Client-Server Paradigm

To show how our generic network specification can be used to model more specific distributed paradigms, we extend it to model the Client-Server paradigm that is used, for example, in CORBA [Object Management Group, 1995]. In this paradigm, certain nodes in the network (often only one) act as a *server*. All other nodes act as a *client*. The clients send requests to a server which runs local code to determine a response to return to the client.

We begin by introducing two more types, *Request* and *Response*,

$$[Request, Response]$$

and two functions, *req* and *resp*, which extract a request or response, respectively, from given data. The functions are partial (denoted by the symbol $\rightarrowtail$) since not all data represents a request or response. In particular, no two pieces of data represent both a request and a response. If we wanted to model sending a request and response in a single message, then an additional partial function could be introduced for this. ($\mathrm{dom}\, f$ denotes the domain of a function f.)

$$\begin{array}{|l}
req : Data \nrightarrow Request \\
resp : Data \nrightarrow Response \\
\hline
\mathrm{dom}\, req \cap \mathrm{dom}\, resp = \varnothing
\end{array}$$

We also introduce a type for programs. A program is a partial function which given a request in its domain returns a response.

$$Program == Request \nrightarrow Response$$

An instance of a program may only be a fragment of code such as a class in an object-oriented program. In such a case, it needs to co-operate with other programs and this would be captured by further constraints.

Given these definitions, we define the class of a server at the top of Figure 2.

Class *Server* inherits class *Node* and adds two new state variables: *programs*, the set of local programs; and *pending*, the set of requests received together with the addresses of the clients who sent them. A set is used for the latter variable, rather than a sequence, to allow for requests to be dealt with in a non-FIFO manner, e.g., according to a priority mechanism, or simply when the required program or a resource it requires is available.

Initially, the set of pending requests is empty. The class has no new operations, but its *Accept* and *Send* operations are extended with new constraints.

If the data field of an accepted message is a request then the data and the source address of the message are added to the set of pending requests. For generality, we assume the data accepted by a server may also be something other than a request. In particular, in a network with more than one server, it may be a response to a request sent by the server in question. When a message which is not a request is accepted, the set of pending requests is unchanged.

If the data field of a sent message is a response then it is being sent to a client who made a request. If this request is in the domain of one of the local programs then the data of the message is the result of applying the local program to the request. Otherwise, the data of the message is left unspecified.

Server

Node

$$\begin{array}{l}
programs : \mathbb{P}\, Program \\
pending : \mathbb{P}(Request \times Address)
\end{array}$$

INIT

$$pending = \varnothing$$

Accept

$$\begin{array}{l}
\Delta(pending) \\
\hline
\textbf{let } m == head\ input\_queue \bullet \\
\quad (m.data \in \operatorname{dom} req \Rightarrow pending' = pending \cup \{(req(m.data), m.source)\}) \land \\
\quad (m.data \notin \operatorname{dom} req \Rightarrow pending' = pending)
\end{array}$$

Send

$$\begin{array}{l}
\Delta(pending) \\
\hline
m!.data \in \operatorname{dom} resp \Rightarrow \\
\quad (\exists\, r : Request;\ a : Address \mid (r, a) \in pending \bullet \\
\qquad m!.destination = a \land ((\exists\, p : programs \bullet r \in \operatorname{dom} p) \Rightarrow m!.data = p(r)) \land \\
\qquad pending' = pending \setminus \{(r, a)\}) \\
m!.data \notin \operatorname{dom} resp \Rightarrow pending' = pending
\end{array}$$

CSNetwork

Network

$$\exists\, n : nodes \bullet n \in \downarrow Server$$

Figure 2: Client-Server network.

This allows further constraints to be added to model particular situations. For example, the response may contain something other than the result of a program such as a message indicating the requested service is unavailable. In all cases, the request is removed from the set of pending requests. For other types of sent messages, the set of pending requests is unchanged.

The class of a Client-Server network is defined at the bottom of Figure 2 as the system class of the Client-Server specification. It is simply an extension of the class *Network* where at least one of the nodes is of class *Server* (or a subclass of *Server*). The power of using inheritance to model incrementally is illustrated well by classes *Server* and *CSNetwork* of Figure 2. The details common to all networks are elided and only those details specific to the Client-Server paradigm are explicitly shown. This aids in both understanding the paradigm and consequently reasoning about its properties.

4 Weak Mobility

In this section, we extend the models of the previous section to model design paradigms supporting weak mobility. These paradigms support the movement of code, but not execution state, about a network.

4.1 The Remote Evaluation Paradigm

The Remote Evaluation paradigm allows a program to be sent along with a request to a server node. It is useful where the client has the program but not the resources needed to execute it. We introduce a subset of data in the domain of *req* which also has a program associated with it (*re_req*). A partial function *prog* returns the program.

$$\begin{array}{l}
re\_req : \mathbb{P}\, Data \\
prog : Data \rightarrow\!\!\!\!\!\!+\ Program \\
\hline
re\_req \subseteq \operatorname{dom} req \\
re\_req \subseteq \operatorname{dom} prog
\end{array}$$

The classes of a node and a server in a Remote Evaluation network are defined at the top of Figure 3. A node is extended with a set of programs. It may send a program with a request which the program can respond to. A server is extended to incorporate these features of a Remote Evaluation node (also via inheritance) and to add received programs to its set of programs. The request is still added to the set of pending requests as specified by the inherited constraints on *Accept* in *Server*. This is necessary as it may not be possible to execute the received program immediately if a resource it requires is being used by another program.

The system class of the Remote Evaluation network specification is defined at the bottom of Figure 3. It is simply a network in which all nodes are of class *RENode* or one of its subclasses, and there exists at least one node of class *REServer*.

4.2 The Code on Demand Paradigm

The Code on Demand paradigm allows nodes to request programs from other nodes. An application of this paradigm is the World Wide Web where browsers routinely download code from remote sites. We introduce a subset of the domain of *resp* corresponding to responses incorporating code (*cod_resp*).

RENode

Node

$programs : \mathbb{P}\, Program$

Send

$m!.data \in re\_req \Rightarrow (\exists\, p : programs \mid req(m!.data) \in \operatorname{dom} p \bullet prog(m!.data) = p)$

REServer

Server

RENode

Accept

$\Delta(programs)$

$$\begin{array}{l}
\mathbf{let}\ m == (head\ input\_queue) \bullet \\
\quad (m.data \in re\_req \Rightarrow programs' = programs \cup prog(m.data)) \wedge \\
\quad (m.data \notin re\_req \Rightarrow programs' = programs)
\end{array}$$

RENetwork

Network

$$\begin{array}{l}
\forall\, n : nodes \bullet n \in \downarrow RENode \\
\exists\, n : nodes \bullet n \in \downarrow REServer
\end{array}$$

Figure 3: Remote Evaluation network.

$cod\_resp : \mathbb{P}\, Data$

$$\begin{array}{l}
cod\_resp \subseteq \operatorname{dom} resp \\
cod\_resp \subseteq \operatorname{dom} prog
\end{array}$$

The class of a node in a Code on Demand network is defined at the top of Figure 4. Note that there is no distinction between client and server nodes in such a network.

The class extends a server with additional constraints on its *Accept* and *Send* operations. When an accepted message is a response incorporating a program then this program is added to those local to the node. When a sent message is a response incorporating a program then this program is a local program of the node.

The system class of the Code on Demand network specification is defined at the bottom of Figure 4. It extends *Network* with the constraint that all nodes are of class *CoDNode* or one of its subclasses.

4.3 Modelling Technologies

The models presented so far allow us to reason about general properties of design paradigms. These models are independent of the technology, i.e., programming language and corresponding run-time support, that might be used to implement them. If we want to consider the effect of adopting a particular technology, we can extend these models with the mechanisms available within that technology. For example, one aspect of mobility that varies depending on the programming language used is that of resource management. The programs local to a node require certain resources. If they are sent to a remote node the availability of these resources must be maintained. This can be done by moving the resources with the program, or by moving a copy of the resources with the program. Alternatively, it can be done by creating a reference to the resources from the remote node, or by simply using similar resources at the remote node.

Java Aglets [Lange, 1997], for example, supports copying the resources and moving them with the program. Given a type *Resource* and a partial function *res* to return the resources associated with appropriate data:

$[Resource]$

$res : Data \nrightarrow Resource$

the Code on Design paradigm using Java Aglets can be captured by an extension to the *CoDNode* class as follows.

CoDNode_JavaAglets

CoDNode

$$\begin{array}{l}
resources : \mathbb{P}\, Resource \\
bindings : Program \nrightarrow \mathbb{P}\, Resource
\end{array}$$

$$\begin{array}{l}
\operatorname{dom} bindings = programs \\
\operatorname{ran} bindings \subseteq resources
\end{array}$$

Send

$$\begin{array}{l}
m!.data \in cod\_resp \Rightarrow \\
\quad res(m!.data) = bindings(prog(m!.data))
\end{array}$$

The other means of resource management could also be captured by extensions to the class *CoDNode*. Similarly, other technology-dependent mechanisms could be incorporated into our models.

5 Strong Mobility

In this section, we look at strong mobility where both code and execution state can move about a

```
CoDNode
  Server
  Accept
    $\Delta(programs)$
    ----
    $\mathbf{let}\ m == head\ input\_queue \bullet$
      $(m.data \in cod\_resp \Rightarrow programs' = programs \cup \{prog(m.data)\}) \land$
      $(m.data \notin cod\_resp \Rightarrow programs' = programs)$
  Send
    $m!.data \in cod\_resp \Rightarrow prog(m!.data) \in programs$
```

```
CoDNetwork
  Network
  ----
    $\forall n : nodes \bullet n \in \downarrow CoDNode$
```

Figure 4: Code on Demand network.

network. In particular, we model the Mobile Agent paradigm supported by languages such as Telescript [White, 1996].

5.1 The Mobile Agent Paradigm

The Mobile Agent paradigm allows code and its execution state to proactively migrate between network nodes[1]. Such code and state are encapsulated in an entity referred to as a *mobile agent*.

We model a mobile agent, or process, as an object in Object-Z. This provides the necessary encapsulation of state and behaviour (i.e., operations). Migration of a process can then be modelled simply as passing the object between nodes representing locations.

The actual state and operations of a process will depend on the application. Here we model just the details common to all processes. Further state variables and operations can be added by inheritance as required.

A process is either active, or migrating. We define a type *Mode* to represent this.

$$Mode ::= active \mid migrating$$

This type is used in class *Process* of Figure 5 to model a process which is initially active and may toggle between being active or migrating. When it moves to migrating mode, it also updates a state variable, *next_address*, denoting the address it wishes to migrate to. The value of this variable will be decided by the process's internal logic which will vary according to the application, and so is left unspecified in class *Process*.

To allow processes to be passed in messages, we introduce a partial function *proc* which given appropriate data returns an object of class *Process*, or one of its subclasses. Such data does not carry a request or response.

```
  $proc : Data \pfun \downarrow Process$
  ----
  $\mathrm{dom}\, proc \cap (\mathrm{dom}\, req \cup \mathrm{dom}\, resp) = \varnothing$
```

[1]A more general definition of the Mobile Agent paradigm would allow both proactive and reactive migration. Here we focus on the former only, although the latter could also easily be incorporated into our model.

The class of a node of a Mobile Agent network is defined at the top of Figure 6. It extends a server with a set of processes. These processes are contained objects, meaning they cannot be associated with more than one node.

The *Accept* operation is extended with constraints captured by an expression involving the operations $Accept_0$ and *AcceptProcess* defined in the class, and the operation *Activate* defined in class *Process*. The operators used are $[]$ ("or") and $\land$ ("and"). $Accept_0$ defines the case when the accepted message does not contain a process, and *AcceptProcess* the case when it does. In the latter case, the process is added to those associated with the node. The conjunction of *AcceptProcess* with the operation *Activate* applied to the accepted process ensures that the process changes mode from migrating to active.

The *Send* operation is constrained so that it only sends local processes which are in migrating mode. These processes are sent to the address of their *next_address* variable, and removed from the set of local processes.

The system class of the Mobile Agent network specification is defined at the bottom of Figure 6. It extends *Network* with a set of processes and a constraint that all nodes are of class *MANode* or one of its subclasses. Initially, all processes are in their initial state, i.e., are active, and the set of processes is exactly those associated with the network's nodes. This is an initial constraint rather than an invariant since, after initialisation, processes may migrate and while being transferred about the network (for example, while in a node's input queue) are not associated with any node.

5.2 Modelling Applications

Given the library of design paradigms in this paper, applications can readily be modelled by adding constraints via inheritance. For example, there are a range of applications that fall into the category of distributed information retrieval. These applications involve gathering information from a set of nodes dispersed about a network. We model here a distributed information retrieval system using the Mobile Agent paradigm (although other paradigms such as Client-Server could also be used).

A process capable of distributed information retrieval is specified by class *DIR_Process* below. Such

Process

$mode : Mode$
$next\_address : Address$

INIT

$mode = active$

Migrate

$\Delta(mode, next\_address)$

$mode = active$
$mode' = migrating$

Activate

$\Delta(mode)$

$mode = migrating$
$mode' = active$

Figure 5: Mobile process.

MANode

Server

$processes : \mathbb{P} \downarrow Process_{\copyright}$

Accept$_0$

$(head\ input\_queue).data \notin \mathrm{dom}\, proc$

AcceptProcess

$\Delta(processes)$

$(head\ input\_queue).data \in \mathrm{dom}\, proc$
$processes' = processes \cup \{proc((head\ input\_queue).data)\}$

$Accept \mathrel{\widehat{=}} Accept_0\ [\!]\ (AcceptProcess \wedge (proc((head\ input\_queue).data)).Activate)$

Send

$\Delta(processes)$

$m!.data \in \mathrm{dom}\, proc \Rightarrow$
$\quad proc(m!.data) \in processes\ \wedge$
$\quad proc(m!.data).mode = migrating \wedge m!.destination = proc(m!.data).next\_address\ \wedge$
$\quad processes' = processes \setminus \{proc(m!.data)\}$
$m!.data \notin \mathrm{dom}\, proc \Rightarrow processes' = processes$

MANetwork

Network

$processes : \mathbb{P} \downarrow Process$

$\forall\, n : nodes \bullet n \in \downarrow MANode$

INIT

$\forall\, p : processes \bullet p.\mathrm{INIT}$
$processes = \bigcup\{n : Nodes \bullet n.processes\}$

$Migrate \mathrel{\widehat{=}} [\, p : processes\,] \bullet p.Migrate$

Figure 6: Mobile Agent network.

a process needs to be able to access information on its host node. Hence, we add a reference to the host node to the state variables of the class *Process*. We also add an itinerary listing the nodes the process must visit, and a set of data representing the information gathered.

Initially, the itinerary is empty and may be updated (to a non-empty itinerary) by the new operation *UpdateItinerary*. A process of this class may only migrate when its itinerary is non-empty, in which case it moves to the address at the head of the itinerary. This address is removed from the itinerary when the process is again activated at its destination. Upon activation its host reference is also updated. After activation the process may gather information with the new operation *GatherInfo* which invokes an operation *ReadData* of the host node.

DIR_Process

Process

$host : \downarrow DIR\_MANode$
$itinerary : \mathrm{seq}\, Address$
$information : \mathbb{P}\, Data$

Init

$itinerary = \langle\,\rangle$

UpdateItinerary

$\Delta(itinerary)$

$itinerary' \neq \langle\,\rangle$

Migrate

$itinerary \neq \langle\,\rangle$
$next\_address' = head\ itinerary$

Activate

$\Delta(host, itinerary)$

$itinerary' = tail\ itinerary$

GatherInfo$_0$

$\Delta(information)$
$data? : \mathbb{P}\, Data$

$information' = information \cup \{data\}$

$GatherInfo \mathrel{\widehat{=}} host.ReadData \parallel GatherInfo_0$

Host nodes of processes of class *DIR_Process* must be of class *DIR_MANode*. This class inherits *MANode* and extends it with a set of local data, and an operation to read an arbitrary subset of the data. It also adds a constraint that all processes associated with a node that are of class *DIR_Process* regard the node as their host. (*self* is an implicitly declared variable in every Object-Z class which for each object of the class is assigned to its reference.) This constraint ensures that a process's *host* variable is updated correctly when the process is activated.

DIR_MANode

MA_Node

$data : \mathbb{P}\, Data$

$\forall\, p : processes \bullet$
$\quad p \in \downarrow DIR\_Process \Rightarrow p.host = self$

ReadData

$data! : \mathbb{P}\, Data$

$data! \subseteq data$

6 Related Work

We are not the first to develop formal models of the design paradigms identified by Fuggetta, Picco and Vigna [Fuggetta et al., 1998].

Picco, Roman and McCann [Picco et al., 1997] develop models of the paradigms using Mobile UNITY. They model, however, a simple application in each of the paradigms, rather than modelling the paradigms independently of an application as we have done. The result, therefore, is a set of guidelines rather than a library of reusable specifications. Although some reuse occurs between the models of the paradigms, this is done in an ad hoc manner rather than by using a language construct such as inheritance. The use of inheritance as a reuse mechanism in our approach, simplifies the specifier's task by focusing it on the differences between models (rather than their usually more voluminous commonalities). It can also aid subsequent reasoning by allowing previously proved properties to be reused (as shown by Smith [Smith, 1995b]).

One technical issue that arises with the Mobile UNITY work is that the components in a specification are fixed. This means programs, which are modelled as components, cannot be copied and hence cannot co-exist at two locations. Therefore, the model of the Remote Evaluation paradigm, for example, requires that the program is "returned" to the client with a response.

Models of the design paradigms have also been developed using an extension of Cellular Automata by Brooks and Orr [Brooks and Orr, 2002]. Although these models are presented independently of an application, they are less general than ours for two reasons. Firstly, they use one dimensional networks of cells (representing nodes) limiting the approach to fixed and linear network topologies. Secondly, due to the lock-step nature by which cells in Cellular Automata evolve, all communication is done via packets. Hence, process migration includes cutting a process into multiple packets which must be reassembled when received. Our approach abstracts away from these lower-level details which are really part of the underlying technology, rather than the design paradigms.

The Cellular Automata approach also has a different goal to ours. It is concerned with detecting *emergent* properties of mobile systems such as network congestion. Such properties arise from interactions of components and cannot generally be deduced from the properties of the components themselves. Our goal, however, is to use component properties to deduce those of the system (as suggested by Smith [Smith, 1995a] and Griffiths [Griffiths, 1997]). Hence, the two approaches can be seen as complementary.

Object-Z is a very general formalism and has found application in a wide variety of areas. It is interesting therefore to also compare our work with formalisms developed specifically for mobile systems.

Most of these are either process algebra approaches, such as the π-calculus [Milner, 1999] and its variants, or stream-based formalisms, such as that of Grosu and Stølen [Grosu and Stølen, 2001]. These formalisms model systems in terms of sequences of events which flow over communications channels. They have no explicit support for modelling the state of system components, and little support for modelling data structures. While they are notationally and semantically elegant, their lack of support for modelling state and data structures makes them less suited to modelling the often complex details of mobile system technologies and applications.

This issue is addressed by Taguchi and Dong [Taguchi and Dong, 2002] who combine Object-Z with a process algebra supporting mobility primitives to produce a formalism called MobiOZ. Our work shows, however, that Object-Z alone is sufficient for modelling mobile systems. Furthermore, the mobility primitives in MobiOZ are based on those in Telescript [White, 1996] and hence the language is not as adaptable to other technologies, including those yet to be developed, as our approach.

7 Conclusion

We have presented a library of Object-Z classes modelling the dominant design paradigms used in mobile systems. We have done this by taking advantage of object-oriented structuring techniques to allow for dynamic network connectivity (connections are potentially mutable references between nodes) and proactive process migration (processes are objects which may be passed as parameters and which may initiate events). We have also used inheritance and polymorphism to specify our models incrementally, exploiting commonalities among the design paradigms. Inheritance can similarly be used to extend our models to represent their realisation with specific technologies, or to capture their use in specific applications.

The formal basis of Object-Z allows us to use the models and their potential extensions to reason about properties of mobile systems. This process can be aided by taking advantage of the substantial reuse of classes within our framework, and the particular form to which our specifications conform. Future work will look at how these aspects of the models can be exploited to simplify reasoning and, hence, the development of suitable reasoning support tools.

Acknowledgement: Preparation of this paper was funded in part by Australian Research Council Linkage grant LP0347620, Formally-Based Security Evaluation Procedures.

References

[Baldi et al., 1997] Baldi, M., Gai, S., and Picco, G. (1997). Exploiting code mobility in decentralized and flexible network management. In [Rothermel and Popescu-Zeletin, 1997].

[Bert et al., 2003] Bert, D., Bowen, J., King, S., and Waldén, M., editors (2003). *3rd International Conference of Z and B Users (ZB 2003)*, volume 2651 of *LNCS*. Springer-Verlag.

[Brooks and Orr, 2002] Brooks, R. and Orr, N. (2002). A model for mobile code using interacting automata. *IEEE Transactions on Mobile Computing*, 1(4):313–326.

[Fuggetta et al., 1998] Fuggetta, A., Picco, G., and Vigna, G. (1998). Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

[Ghezzi and Vigna, 1997] Ghezzi, C. and Vigna, G. (1997). Mobile code paradigms and technologies: A case study. In [Rothermel and Popescu-Zeletin, 1997], pages 39–49.

[Griffiths, 1997] Griffiths, A. (1997). Modular reasoning in Object-Z. In Wong, W. and Leung, K., editors, *Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference*, pages 140–149. IEEE Computer Society Press.

[Grosu and Stølen, 2001] Grosu, R. and Stølen, K. (2001). Stream based specification of mobile systems. *Formal Aspects of Computing*, 13(1):1–31.

[Kassel and Smith, 2001] Kassel, G. and Smith, G. (2001). Model checking Object-Z classes: Some experiments with FDR. In *Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 445–452. IEEE Computer Society Press.

[Lange, 1997] Lange, D. (1997). Java Aglets Application Programming Interface (J-AAPI). White paper, IBM Corp.

[McComb and Smith, 2003] McComb, T. and Smith, G. (2003). Animation of Object-Z specifications using a Z animator. In *International Conference on Software Engineering and Formal Methods (SEFM 2003)*. IEEE Computer Society Press.

[Milner, 1999] Milner, R. (1999). *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press.

[Object Management Group, 1995] Object Management Group (1995). *CORBA: Architecture and Specification.*

[Picco et al., 1997] Picco, G., Roman, G.-C., and McCann, P. (1997). Expressing code mobility in Mobile UNITY. In Jazayeri, M. and Schauer, H., editors, *6th European Software Engineering Conference (ESEC'97)*, volume 1301 of *LNCS*, pages 500–518. Springer-Verlag.

[Rothermel and Popescu-Zeletin, 1997] Rothermel, K. and Popescu-Zeletin, R., editors (1997). *First International Workshop on Mobile Agents (MA'97)*, volume 1219 of *LNCS*. Springer-Verlag.

[Smith, 1995a] Smith, G. (1995a). Extending $\mathcal{W}$ for Object-Z. In Bowen, J. and Hinchey, M., editors, *9th International Conference of Z Users*, volume 967 of *LNCS*, pages 276–295. Springer-Verlag.

[Smith, 1995b] Smith, G. (1995b). Reasoning about Object-Z specifications. In *Asia-Pacific Software Engineering Conference (APSEC95)*, pages 489–497. IEEE Computer Society Press.

[Smith, 2000] Smith, G. (2000). *The Object-Z Specification Language.* Advances in Formal Methods. Kluwer Academic Publishers.

[Smith et al., 2002] Smith, G., Kammüller, F., and Santen, T. (2002). Encoding Object-Z in Isabelle/HOL. In Bert, D., Bowen, J., Henson, M., and Robinson, K., editors, *International Conference of Z and B Users (ZB 2002)*, volume 2272 of *LNCS*, pages 82–99. Springer-Verlag.

[Smith and Winter, 2003] Smith, G. and Winter, K. (2003). Proving temporal properties of Z specificatons using abstraction. In [Bert et al., 2003], pages 260–279.

[Spivey, 1992] Spivey, J. (1992). *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition.

[Taguchi and Dong, 2002] Taguchi, K. and Dong, J. (2002). An overview of Mobile Object-Z. In George, C. and Miao, H., editors, *4th International Conference on Formal Engineering Methods (ICFEM 2002)*, volume 2495 of *LNCS*, pages 144–155. Springer-Verlag.

[White, 1996] White, J. (1996). Telescript technology: Mobile agents. In Bradshaw, J., editor, *Software Agents*, pages 437–472. AAAI Press/MIT Press.

[Winter and Smith, 2003] Winter, K. and Smith, G. (2003). Compositional verification for Object-Z. In [Bert et al., 2003], pages 280–299.