Specifying distributed system services

L B Arief, M C Little, S K Shrivastava, N A Speirs and S M Wheater

Building a software service requires careful analysis of the requirements given by the customer for the system. It is often difficult to understand the system requirements correctly, due to the fact that they are usually described in plain language. This difficulty could be overcome if a sufficiently precise description of system services to be provided can be produced that is easy to follow by both customers and designers. Given such a specification of a service that on the surface permits several ways of implementing it, the design team should be able to select with reasonable confidence, the most appropriate set of design options, before commencing the building of the service. Naturally, this requires the development of modelling and analysis techniques that enable the evaluation of various design options for a given service. As a first step towards achieving these goals, the paper briefly reviews current approaches to specifying system architectures and explores the suitability of the Unified Modelling Language (UML) as a specification tool.

1. Introduction

Building a software system requires careful planning and investigation in order to avoid any problems in the later stages of the development. The first thing to do is to analyse the requirements of system services given by the customer. It is often difficult to understand the requirements correctly, due to the fact that they are usually described in plain language. What we require is a description of system services that is sufficiently precise to be easily followed by both customers and designers. Given such a specification of a service that on the surface permits several ways of designing it, system implementors should be able to select, with reasonable confidence, the most appropriate set of design options, before commencing the building of the service.

A problem that one encounters right away is how to express operating conditions and service requirements in a manner that permits construction of a model reflecting those conditions and requirements. Naturally, this requires developing modelling and analysis techniques that enable evaluation of various design options for a given service architecture.

As a first step towards achieving these goals, the paper briefly reviews current approaches to specifying system architectures and explores the suitability of the Unified Modelling Language (UML) as a specification tool that can be used for developing simulation models of distributed systems and services. UML was chosen as it has been adopted by the Object Management Group (OMG) as a standard and is increasingly being used in industry.

2. Architecture description languages

Software architecture specification is intended to describe the structure of the components of a software system, their interrelationships, and principles and guidelines governing their design and evolution; a component is defined here as a (distributed) self-contained object (computational unit). Work in this area has produced high-level notations (architecture description languages — ADLs) for expressing and representing architectural designs and styles. This section will review work from a representative set of research groups — Darwin [1], UniCon [2] and Rapide [3, 4].

2.1 Darwin and UniCon

Many of the current generation of ADLs are based upon specifying distributed systems in terms of components and connectors. Components are compilation units, objects, etc, of various types which are specified by interfaces. Components interact via connectors, which are specified by protocols; a connector is responsible for mediating interactions between components, i.e. they define and impose rules governing the interactions between components.

Components can be either primitive or composite, and can be connected by multiple connectors. Connectors can be simple procedure calls, remote procedure calls, Unix pipes, files, etc, and can connect multiple components.

A well-known system for specifying system requirements is Darwin [1, 5], which is the configuration language for the Regis programming environment [6]. In Darwin, components are strongly typed first-class language primitives, supporting single inheritance. A component interface specifies what the component can provide to others, and what it requires. These 'provide and require' statements are implicitly in terms of connectors. For example, consider Fig 1, which shows a filter component, which provides communication object 'left', and requires communication objects 'right' and 'output'.

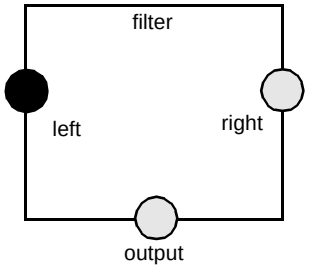

Fig 1 Filter component.

In Darwin, the specification for this component would be[1]:

```
component filter {
  provide left <port, int>;
  require right <port, int>,
          output <port, int>;
}
```

Darwin supports a bind statement which is used to tie together components using their 'provide and require' statements. The Darwin compiler checks that connections are only made between compatible communication objects. For example, the specification for a chain of pipelined filter components as depicted in Fig 2 would be as given below.

[1] Ports are one of the standard communication classes provided by Darwin.

```
component pipeline {
        provide input;
        require output;

        Filter1 filter;
        Filter2 filter;

    inst Filter1;
    inst Filter2;

  bind input - - Filter1.left;
  bind Filter1.output - - output;
  bind Filter1.right - - Filter2.left;
  bind Filter2.output - - output;
  bind Filter2.right - - output;
}
```

Although connections are implicitly specified using the 'provide and require' statements, there is no explicit connector language construct. Darwin considers the component abstraction powerful enough to encompass connectors, i.e. if a specific type of connector is required, it can be specified as a component, with other components connected to it. For example, a Unix pipe 'connector' could be specified in Darwin as:

```
component UnixPipe {
    provide source <port line>;
    require sink <port line>;
    require error <port line>;
    bind source - - sink;
}
```

UniCon is an architecture description language developed at Carnegie Mellon University [2, 7]. In UniCon an interface consists of the component's type, specific properties that specialise the type, and a list of players (methods) through which the component can interact with the rest of the system. Players are also typed entities, and may declare properties which further specify the player. UniCon has a much richer set of language constructs than Darwin for specifying connectors — whereas connections are not first-class objects in Darwin, they are in UniCon. Programmers have more control of the types of connectors and the roles they play. Although early versions of UniCon

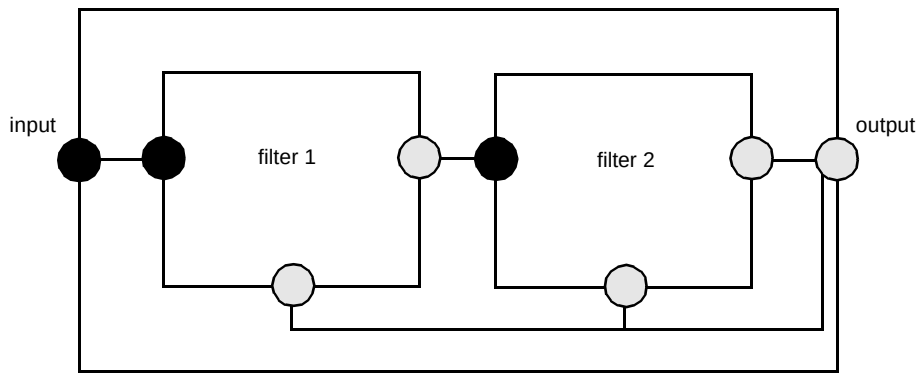

Fig 2 Pipelined filters.

only supported built-in connectors, the aim of the designers is to allow programmers to specify and refine connectors to better suit their requirements.

The main difference between languages such as Darwin and UniCon is whether or not connectors should be treated as first-class objects. In UniCon there is a specific language mapping for connectors, unlike in Darwin. For example, the Unix pipe example above would be specified in UniCon as:

```
CONNECTOR UnixPipe
  PROTOCOL IS
    TYPE Pipe
    ROLE source IS source
         MAXCONNS(1)
         END source
    ROLE sink IS sink
         MAXCONNS(1)
         END sink
    ROLE err IS sink
         MAXCONNS(1)
         END err
    END PROTOCOL
  IMPLEMENTATION IS
    BUILTIN
  END IMPLEMENTATION
END UnixPipe
```

There is much debate over whether connectors should be explicit objects within the language as they are in UniCon, or implicit as they are in Darwin [2, 8].

2.2 Rapide

Rapide is an event-based, concurrent object-oriented language specifically designed for prototyping architectures of distributed systems. It goes beyond the work of Darwin and UniCon in concentrating more on the specification of software modules, what they provide and require from other modules, and how their combination achieves an architectural specification. The Rapide language is accompanied by a variety of tools to aid in the specification, design, and testing of software modules and architecture. Because the language is object-oriented, the distinction between a module and an architecture depends upon the context in which it is used, i.e. an architecture can be a module in another application.

Design goals

The primary design goals of Rapide are:

- to provide architecture constructs that permit systems to be expressed in a suitable form for simulation before implementation decisions are made,
- to adopt an execution model which captures distributed behaviour and timing,
- to provide formal constraints and mappings to support constraint-based definition of reference architectures and testing systems for conformance to architecture standards,
- to address issues of scalability.

An architecture is essentially a template for a suite of systems (possible implementations of the architecture), consisting of a set of specifications of modules (interfaces), a set of connection rules that define communication between interfaces, and a set of formal constraints that define legal and/or illegal patterns of communication. As with Darwin/UniCon, an interface defines the features provided to, and required from, other modules. An interface can have an abstract definition of the behaviour of modules. Typically such behaviour specifies relationships between data received and data generated by a module. Formal constraints specify restrictions on various aspects of interfaces and connections, such as relations between data, timing constraints, etc.

Architectures are based upon an event-based execution model — poset (partially ordered set of events). Interface behaviours execute by waiting to receive events and then reacting by generating new events. Connections define how events generated by one interface cause other events to be received by another interface. Constraints place restrictions on event activity, both in interfaces and over the set of connections. Importantly, constraints are checkable, and Rapide includes a suite of development tools to allow architecture implementations to be verified with respect to architecture descriptions at the event level. These tools allow the gradual instantiation of an architecture, module by module, into a final system. At each stage of system development, the current implementation can be simulated and verified for conformance to its architecture (e.g. type checking and constraint verification).

Rapide environment

Rapide consists of five parts, which are briefly described in the following sections:

- the types language for describing the interfaces of components,
- the architecture language for describing the flow of events between components,
- the specification language for writing abstract constraints on the behaviour of components,
- the executable language for writing executable modules,
- the pattern language for describing patterns of events.

Types language

A component consists of two parts — an interface which defines those features through which it interacts with other components, and a module that either encapsulates an executable prototype of the component, or hierarchically defines the component as an architecture of other components. Interfaces are defined in the types language, whereas modules are either defined in the executable language or the architecture language. The types language supports object-oriented and abstract data type styles of defining interfaces, and supports multiple interface inheritance.

The interface below is a simple null-filter — whenever a receive event occurs, the module which implements the interface will generate a send event with the same parameter:

```
type Application is interface
     extern action Request(p : params);
     public action Results(p : params);
behavior
     (?M in String) Receive(?M) =>
Results(?M);;
```

Architecture language

An architecture declares a set of components and a set of connections between extern and public constituents of interfaces of components. As mentioned previously, an architecture is also a module with its own interface. As a result of a connection, events generated by one module cause events to be received by another, or functions called by one module are executed by another. Therefore, an architecture defines dataflow and synchronisation between modules. An example architecture of two of the null-filters above could be:

```
architecture Example is
    P : Application; Q : Application;
connect (?M in String)
    P.Results(?M) to Q.Request(?M);
end Example;
```

The connect statement can be arbitrarily complex, for example, specifying timing constraints, state constraints.

Executable language

The executable language provides concurrent, reactive programming constructs for writing modules. Modules are defined by a set of processes that observe and react to events by executing arbitrary code that may generate new events.

Pattern language

Rapide has an extensive set of language constructs for specifying patterns of events; only two representative examples are considered here:

- dependent: $P \rightarrow P'$ — a match of patterns P and P' where all events which matched P' depend on all events which matched P,
- temporal restriction: P during(m, n) — a match for P where all the events of the match start and finish within the time interval m to n, inclusive.

Specification language

The specification language uses a combination of algebraic and pattern constraints. A constraint placed in an interface constrains visible executions of modules of the type. Constraints can also be placed in architectures. Interface constraints include constraints on parameter values of the interface functions and actions, algebraic constraints on the abstract state of the interface, and pattern constraints on the events that can be generated and observed from the interface. An interface constraint therefore represents a contract specifying how to use a module of that type, and what a module of that type promises to do. For example:

```
type Application is interface
   public action Receive(Msg : String);
   extern action Results(Msg : String);
constraint
   match
     ((?S in String)(Receive(?S) ->
             Results(?S)))^(*~);
end Application;
```

The constraint specifies that all messages taken in are delivered. Before a `Results` event can occur, there must have been a corresponding `Receive` event (with the same parameter), and there can be any number of these events (*). Furthermore, these pairings of events must be disjoint, i.e. unrelated (~).

2.3 Summary

As the brief description of Rapide has shown, the language and environment are extremely rich in the capabilities required to specify arbitrary, complex distributed systems. Unlike Darwin and UniCon where specification of connections between components is seen as the important goal, this is only a part of the architectural specification which Rapide addresses. For example, specification of timing constraints on operations by users of components can be verified with respect to timing constraints guaranteed by those components.

3. Unified Modelling Language

UML is (mainly) a set of graphical notations for specifying, visualising, constructing and documenting the artefacts of software systems [9]. The design of a system is captured in UML using a set of graphical notations:

- class diagrams — show the static structure of the system, i.e. the types of object within the system and the static relationships that exist between them,
- use case diagrams — show interactions between users and the computer system,
- interaction diagrams — show the pattern of interactions between objects in a system; there are two types of interaction diagram:

  — sequence diagrams arrange the interactions in time sequence,

  — collaboration diagrams show the interactions in terms of links between the objects,
- state diagrams — describe the possible states a particular object can get into and how the object's state changes as a result of events,
- activity diagrams — represent the activities that are triggered at the completion of an operation,
- implementation diagrams — show aspects of implementation, including the structures of the implementation and the source codes.

Each type of diagram provides a different view of the system, thus helping to reduce the complexity of individual diagrams. For example, a class diagram can be used to describe the static properties of the system while the activity diagram is used to show the interactions that can happen in the system.

As it is essential to understand the purposes of these diagrams, some explanation of the most commonly used diagrams is provided below [9].

- The class diagram

  The class diagram is a graph that classifies the static elements (classes) of the system and shows the static relationships between these elements. A class represents a set of objects with similar structure (attributes), behaviour (operations) and relationships. The relationship between two (or more) classes can be as shown below.

  — Association indicates what role one class has in the relationship. There are some additional notations available, such as the multiplicities (which indicate how many instances of one class can take part in the association), aggregation (to show that one class is a collection of several instances of another class), composition (one class is a part of another class) and dependency (to indicate that one class depends on another).

  — Generalisation is used to capture the notion of inheritance and shows the relationship between a more general element (the supertype) and a more specific element (the subtype). The subtype inherits the property of its supertype, i.e. the subtype must be consistent with the supertype, and it may add some additional (more specific) information.

  The notations for the class diagram are depicted in Fig 3.

- The sequence diagram

  The pattern of interaction between objects can be shown by using one of the two interaction diagrams — the sequence diagram or the collaboration diagram. In this paper, only the sequence diagram will be discussed further.

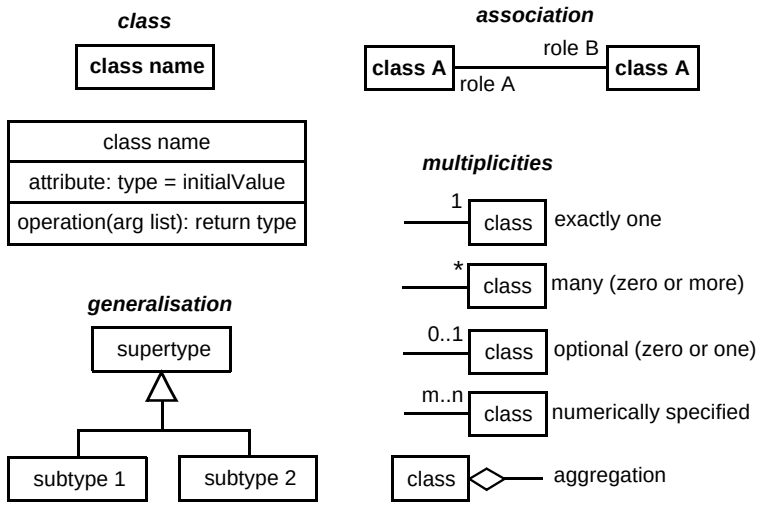

Fig 3 Class diagram notations.

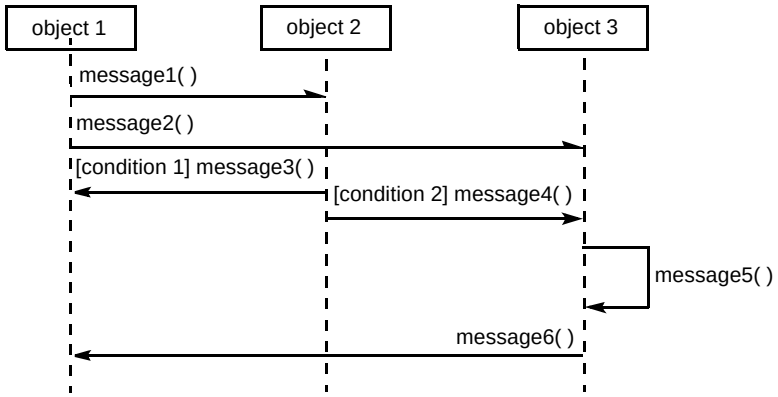

Fig 4 Sequence diagram notations.

A sequence diagram shows an interaction of the objects in the system in a time sequence (see Fig 4). An interaction consists of messages that are exchanged between objects in order to obtain the desired result of an operation. In the sequence diagram, the objects are arranged horizontally (on top of the diagram) while the progress of time is shown by a vertical bar (normally proceeding down the page), one for each object. A message is represented by an arrow between the timelines of the objects, although, on some occasions, an arrow can point to its own timeline to indicate self-invocation. An asynchronous message is represented by a half-arrowhead. Conditions can also be included to indicate that the message is only sent if the conditions are satisfied (see Fig 4).

- The activity diagram

An activity diagram shows operations that can be performed in order to achieve a certain goal, and the transitions triggered by the completion of the operations (Fig 5). An activity diagram is a special case of the state diagram [9].

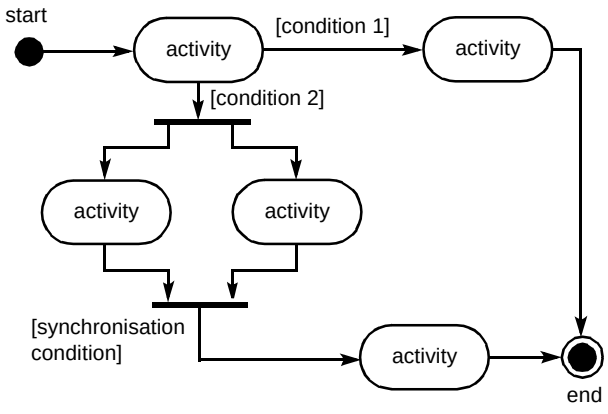

Fig 5 Activity diagram notations.

- Other notations

There are some additional notations that are useful (see Fig 6) — the constraint notation (such as some condition that must be maintained), the note notation (for displaying textual information, such as comments) and the object notation (which is also used in the sequence diagram).

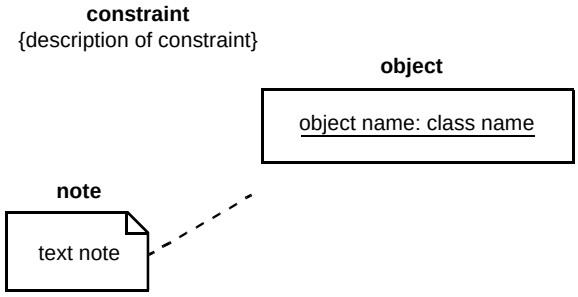

Fig 6 Other useful notations.

The UML offers facilities for system description that appear comparable to Rapide. However, as yet there are no analysis and simulation tools available to match Rapide's facilities. This situation is expected to change as the language is gaining in popularity and is increasingly being used in industry. Section 5 describes the development of a UML simulation tool.

4. Example — specification of an IN application

New features for call handling (e.g. credit call charging, 0800 calls) in intelligent networks (INs) are typically delegated to computer systems that are attached to switches. A switch passes the incoming call request requiring special processing to the local computer system for processing before setting up the connection. The processing must be performed fairly quickly (less than a second for most of the calls). These computer systems maintain data pertaining to the customers. Two operations provided for each customer object are makeCall and receiveCall; the former maintains information relating to outgoing calls (e.g. should the call be barred) and the latter maintains information concerning incoming calls (e.g. does the receiver wish to receive calls from the caller).

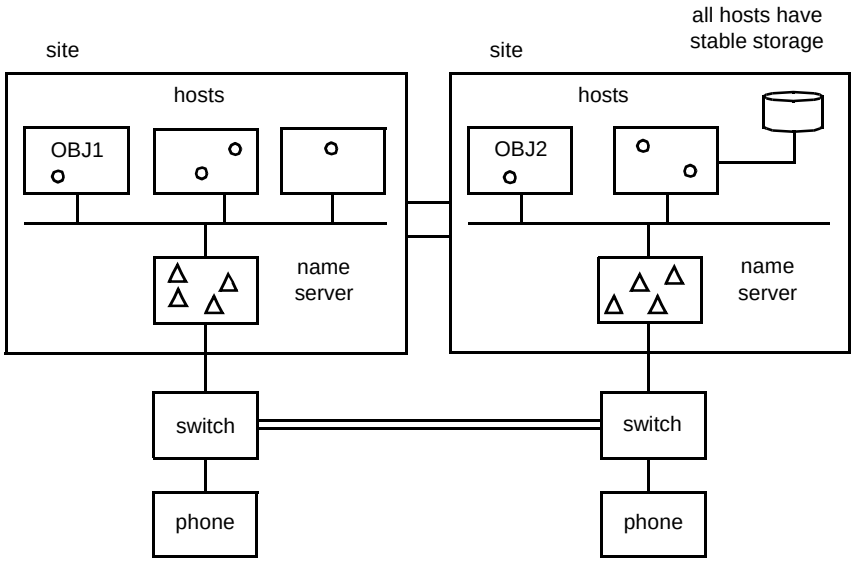

Fig 7 The architecture of the new system.

4.1 Physical architecture

The basic system structure is shown in Fig 7. Processing at a switch site is performed by a high-performance computing cluster, comprising of about ten hosts.

There are between 60 and 1000 sites. Each host has 10—100 Gbit memory. Sites are connected by WANs with bandwidth of ~34 Mbit/s and a latency of ~50 ms. The total number of customer objects in the system is in the range of 10^5—10^8. Communication within a site is via a LAN with 100 Mbit/s bandwidth and a latency of ~1 ms.

4.2 Processing requirements

The processing requirements are given, together with the steps involved in making a call. Processing is initiated via messages from a switch which contain two parameters — the calling line identity (CLI) and the dialled number (DN). Between 3000 to 100 000 messages per second are to be handled. Each message is first handled by a name server that maps CLI and DN to corresponding customer object identifiers, OBJ1 and OBJ2 respectively.

These are unique identifiers which contain the address and host number of the appropriate makeCall and receiveCall objects.

OBJ1.makeCall(OBJ2) is then invoked to perform some caller-specific processing; in particular, makeCall checks to see that the barOutgoing flag is not set and then makes RPC to OBJ2.receiveCall(OBJ1). receiveCall checks the blacklist for OBJ1 and sends back a startRinging reply. makeCall must service 90% of calls in at most 500 ms, 95% of calls in at most 5000 ms, and 100% of calls in at most 10 000 ms.

4.3 Specification using UML

The elements of the architecture and the requirements above can be translated into several UML diagrams.

- The physical architecture

  The main details of the underlying hardware (the machines and the connection between them) and application objects can be represented as a class diagram (see Fig 8).

- Application logic

  Customers can only make a call if they are not barred from doing so by the telephone company (e.g. due to unpaid bills). Another requirement is that the caller is not on the blacklist of the called party. The operation makeCall tries to connect two customers through a series of operations that involves several checks to ensure that both requirements are satisfied. As a customer can have more than one phone number (in different locations), it is necessary that makeCall accesses the name server to find the latest binding indicating where the called party can be reached.

Figure 9 shows the sequence diagram which represents the makeCall operation. This diagram also indicates the time constraints of the makeCall operation. It should be noted that at present UML cannot describe the probabilistic constraints except in comments or notes.

In addition to the sequence diagram, an activity diagram can also be used to show a possible scenario when a customer tries to telephone another customer using this system (Fig 10).

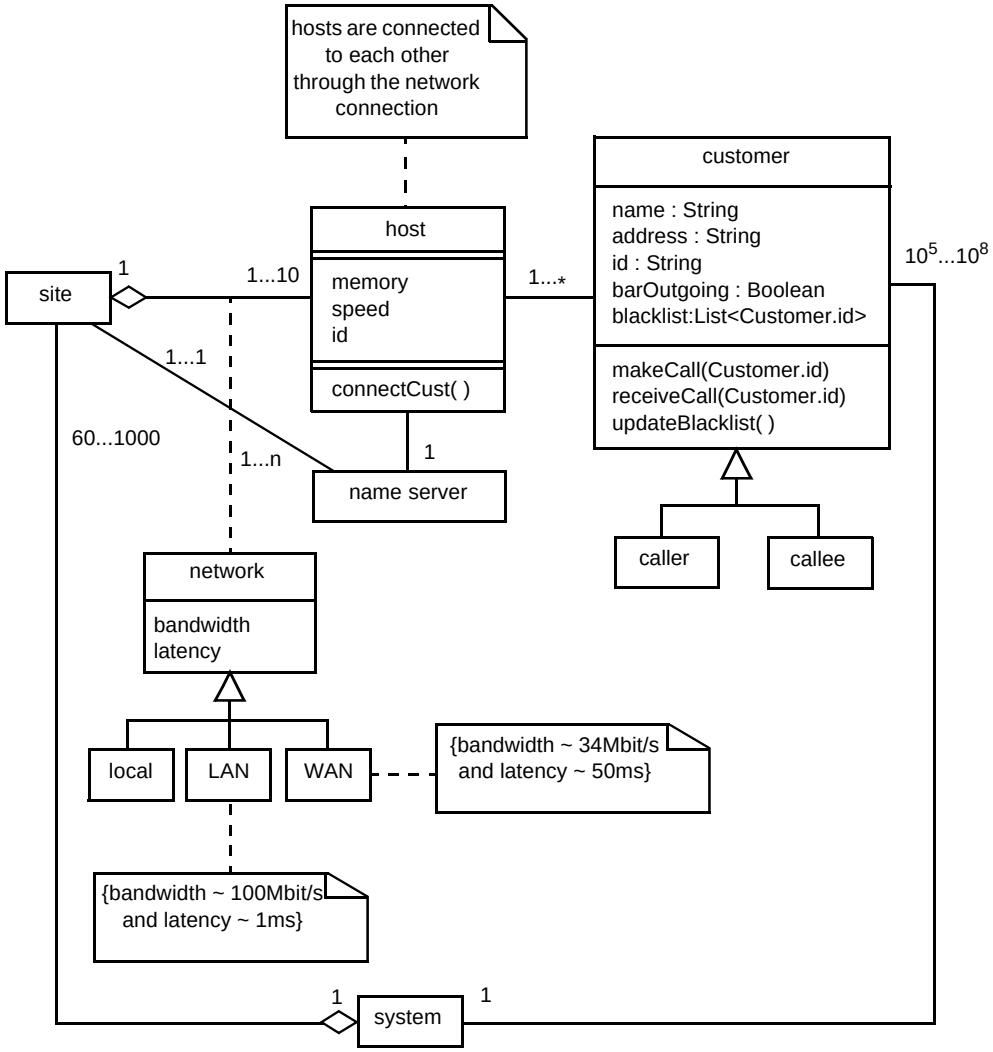

Fig 8 Class diagram.

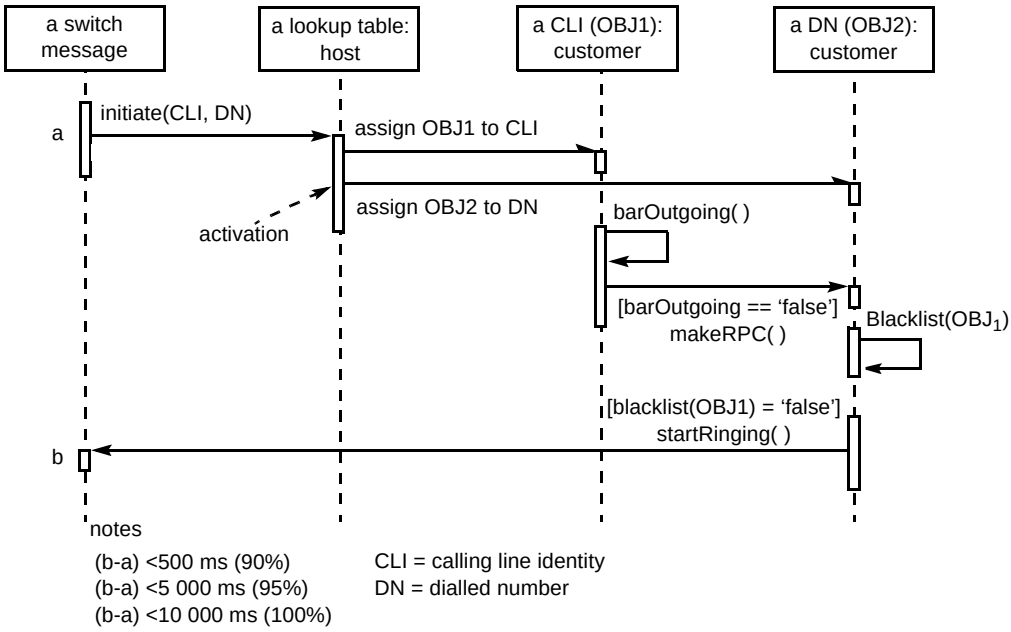

Fig 9 Sequence diagram.

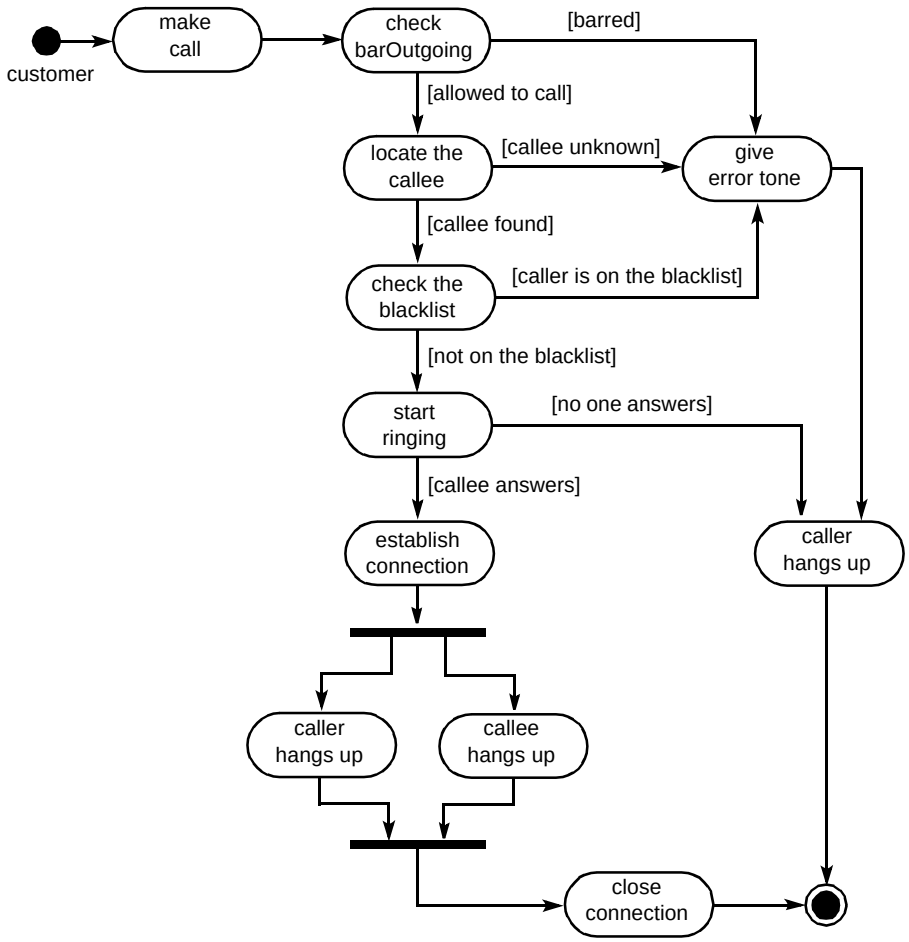

Fig 10 Activity diagram.

5. Automatic generation of simulation models

UML design tools are becoming available to make the job of drawing the UML diagrams easier. The best known is the Rational Rose tool [10], which has the following features:

- it provides the standard UML shapes, so there is no need to create one's own,
- the diagrams for a particular specification can be saved into one model, which improves the organisation of the graphs, and, within one model, the user can view different diagrams and browse through them by expanding these diagrams into their components,
- different views are provided — use case view, logical view, component view and deployment view,
- documentation can be attached to each diagram — these documents can help in understanding the meaning of the diagrams, as they can contain the original specification or other useful comments,
- as each component of the diagram is an object, it is possible to change them separately and to relate them to other (appropriate) components in the diagram,
- a report and C++ skeleton codes can be automatically generated from the diagrams.

Although such tool sets are quite useful for drawing UML diagrams, they can be made even more useful if they can be augmented with modelling and analysis tools (e.g. a simulation tool) based on the facts and constraints given in the diagrams, in order that the performance of the proposed system can be analysed.

Current development is aimed at producing such a tool which could be used to generate a simulation model from a design specification automatically without the requirement for users to know how to construct simulation programs. The tool is intended to be suitable for a variety of distributed system applications and would allow different performance requirements to be incorporated in the simulation. In order to achieve this, the generic components of a simulation model have begun to be investigated. By being able to identify these components, the construction of a simulation program can be quite a straightforward process as it only involves the composition of the components in an appropriate way. As, inevitably, there may still be some special components required for a particular simulation,

there is a need to provide room for specific simulation requirements to be incorporated into a simulation.

In the current implementation, a textual UML description of the problem is fed to the simulation generator which produces code for use with the C++SIM tool-kit [11]. The code produced can be immediately compiled and run. The simulation generator currently produces only simple simulations where work is generated by a single process and is queued for servicing by a second processor. The tool is currently being extended to allow simulation of a pipeline of services to be generated directly from UML.

It is expected that simulations of problems similar to that specified previously may soon be generated automatically from their UML representation.

6. Conclusions

The aim was to investigate easy-to-use, but sufficiently precise, software architecture specification techniques that can describe the structure of the components of a software system, their interrelationships, and principles and guidelines governing their design. In addition, given the specification of a service that on the surface permits several ways of designing it, one should be able to select, with reasonable confidence, the most appropriate set of design options, before commencing the building of the service. The paper briefly reviewed current approaches to specifying system architectures and examined the suitability of the Unified Modelling Language in more detail. UML has been adopted by the Object Management Group as a standard and is increasingly being used in industry as a specification tool.

UML turns out to be quite useful for this project. By translating the requirements into several diagrams, different views of the problem can be obtained and the requirements can be made more formal. UML design tools are becoming available to make the job of drawing the UML diagrams easier.

Although such tool-sets are quite useful for drawing UML diagrams, they can be made even more useful if they can be augmented with modelling and analysis tools (e.g. the simulation tool described earlier), based on the facts and constraints given in the diagrams, so that the performance of the proposed system can be analysed.

Acknowledgements

This work has been supported in part by a research grant from BT, and in the case of L B Arief, a PhD studentship funded by Newcastle University, Department of Computing Science. Paul Martin, BT Laboratories, supplied the IN application example.

References

1 Magee J et al: 'A constructive development environment for parallel and distributed programs', Proceedings of IEEE 2nd International Workshop on Configurable Distributed Systems, pp 4—14 (March 1994).

2 Shaw M et al: 'Abstractions for software architecture and tools to support them', IEEE Transactions on Software Engineering, 21, No 4, pp 314—335 (April 1995).

3 Luckham D C et al: 'Specification and analysis of system architecture using Rapide', IEEE Transactions on Software Engineering, 21, No 4, pp 336—355 (April 1995).

4 Luckham D C et al: 'An event-based architecture definition language', IEEE Transactions on Software Engineering, 21, No 4, pp 717—734 (April 1995).

5 Magee J et al: 'Structuring parallel and distributed programs', IEEE Software Engineering Journal, 8, No 2, pp 73—82 (March 1993).

6 Kramer J and Magee J: 'Dynamic configuration for distributed systems', IEEE Transactions on Software Engineering, SE-11, No 4, pp 425—436 (April 1985).

7 Shaw M et al: 'Abstractions and implementations for architectural connections', Proceedings of IEEE 3rd International Conference on Configurable Distributed Systems, pp 2—10 (May 1996).

8 Bishop J and Faria R: 'Connectors in configuration programming languages: are they necessary?', Proceedings of IEEE 3rd International Conference on Configurable Distributed Systems, pp 11—18 (May 1996).

9 Fowler M and Scott K: 'UML distilled: applying the Standard Object Modelling Language', Addison-Wesley (1997).

10 Rational Rose — http://www.rational.com/uml/documentation.html

11 Little M C and McCue D L: 'Construction and use of a simulation package in C++', C User's Journal, 12, No 3 (March 1994).

Leonardus B Arief received his BSc in Computing Science from the University of Newcastle upon Tyne in 1997.

He is currently a PhD student at Newcastle University with a scholarship from the Department of Computing Science.

His research interests include distributed system, simulation, specification languages, and automatic code generation from software specification.

Mark C Little received his BSc in Physics and Computing Science from Newcastle University in 1987 and a PhD in Computing Science in 1991.

Since 1990 he has been on the research staff of the Department of Computing Science at Newcastle where he is currently a Senior Research Associate. He is one of the principal designers and implementors of the Arjuna reliable distributed programming system, and has been working on a standards-compliant version of this system since 1997.

His research interests include reliable distributed computing, object-oriented programming languages, object replication, and operating systems.

Santosh Shrivastava obtained his PhD in Computing Science at Cambridge in 1975. Before that he worked for several years in industry. After gaining his PhD, he joined the Department of Computing Science at the University of Newcastle upon Tyne, where his present position is Professor of Computing Science. He leads the Department's research group on distributed computing. His main areas of interest are in the field of fault-tolerant computing. His group has built the Arjuna object-oriented fault-tolerant distributed transaction system. Current research work is focused on building a transactional workflow toolkit for reliable computing over the Internet.

Neil Speirs obtained a 1st class Honours degree in Mathematics from the University of Newcastle upon Tyne in 1980 and a doctorate in Theoretical Physics from the University of Durham in 1985. For two years he worked for Sagesoft Ltd writing many commercial packages. He then worked for Mari Applied Microelectronics Ltd where he was project leader on the Esprit Projects Concordia and Delta-4, both of which were concerned with the design and implementation of fault-tolerant distributed computer systems. Since 1987, he has been a lecturer in Computing Science at the University of Newcastle upon Tyne. His main research interests are in fault-tolerance, reliability and distributed systems. He was the deputy project manager on the Esprit Delta-4 project. He has since led the implementation effort on Voltan — a project to build fail-controlled computing nodes using off-the-shelf components.

Stuart Wheater received his BSc in Computing Science, from the University of Newcastle upon Tyne in 1985 and a PhD entitled 'Constructing Reliable Distributed Applications using Actions and Objects', from the University of Newcastle upon Tyne in 1990. He is currently a Senior Research Associate in the Department of computing Science at Newcastle, where he is a member of the team developing the Arjuna reliable distributed programming system. His research interests include distributed computing, reliable systems, object-oriented computing and tools. His recent work has been in the area of reliable workflow systems.