The A1✓ Architecture Model

Andrew Berry and Kerry Raymond

CRC for Distributed Systems Technology, The University of Queensland, Australia

The A1✓ (the "✓" is silent in speech) model provides the basis for development of distributed applications and distributed infrastructure within the CRC for Distributed Systems Technology. The model has been developed to overcome some of the deficiencies in the RM-ODP standard, in particular, to address the diverse requirements of participants in the CRC. This paper introduces the major concepts of the A1✓ model, and discusses both its relationship with RM-ODP and how it overcomes deficiencies identified in RM-ODP.

Keyword Codes: C.2.4; D.2.10
Keywords: Distributed Systems; Design

1. INTRODUCTION

The A1✓ model has been created as a basis for development of distributed applications and development of an infrastructure to support these applications. The A1✓ model underpins the work of the CRC for Distributed Systems Technology (the DSTC), a consortium of research, industrial and end-user organisations in Australia. The model defines a minimal set of concepts and rules for distributed systems.

The involvement of the DSTC in the ISO/ITU-T RM-ODP standardisation effort [2],[3] has a significant influence on the A1✓ model, noting that the DSTC also has a substantial influence on the RM-ODP standard. Although there are similarities between our work and RM-ODP, the A1✓ model addresses some of the deficiencies in RM-ODP that become evident in an organisation with such diverse interests as the DSTC. In particular:

- the RM-ODP viewpoints are effective for describing systems, but less effective for software engineering;
- the current RM-ODP computational model is inelegant when applied to complex applications. RM-ODP was originally created with RPC and client-server applications in mind, and is not flexible enough to accommodate legacy systems, which is an essential requirement within the DSTC.
- the computational type rules of RM-ODP are too prescriptive;
- the engineering language of RM-ODP prescribes unnecessary structuring detail;
- RM-ODP does not distinguish fundamental infrastructure functionality from other, merely desirable, functionality;

This paper introduces the major concepts of the A1✓ model, and discusses how it overcomes the deficiencies identified in RM-ODP. The complete A1✓ model is described in [1].

Section 2 identifies the important goals of the architecture. Section 3 of this paper describes our development model for distributed systems, identifying two component models, a specification model and an infrastructure model. Sections 4 and 5 describe the specification and infrastructure models respectively. Section 6 describes a number of approaches to the transformation

of a specification to an implementation. Section 7 discusses the relationship of our work to RM-ODP and future plans for the A1✓ model. Section 8 concludes the paper.

2. PHILOSOPHY AND GOALS

The A1✓ model considers only the aspects of distributed systems related to distribution. For openness, no model for data, objects or execution should be imposed, provided that model can co-exist with the constraints required to support distribution of applications across heterogeneous platforms. Therefore, the model primarily discusses the concepts needed to describe the creation, deletion and interaction between entities of a distributed system.

2.1. Openness

The A1✓ model can describe both open or closed (proprietary) implementations. Support for open implementations will, in general, impose some constraints on the model. Since the term "openness" has many interpretations, we see openness as implying the following properties:

- interworking
  The model must allow interworking between heterogeneous software systems. However, we cannot guarantee interworking, since it is only possible to achieve interworking by conformance with interworking standards (both *de-facto* and *de-jure*), or by supporting conversions between systems. The aim is to facilitate interworking, but there is no commitment to total interworking,
- portability
  It should be possible to build applications that are portable across implementations supporting the model. However, portability requires that standards (for example, application programming interface standards) are adhered to, hence some applications might not be portable. Again, the aim is to facilitate portability, but there is no commitment to total portability.
- technology independence
  The model should be independent of any vendor-specific technology, meaning that adopters of the model are not unduly constrained in their choice of vendor.
- autonomy of administration and ownership
  The model must be capable of modelling systems with multiple autonomous administrations and diverse ownership of resources.

Note that these goals are often conflicting, for example, a way to ensure portability might be to buy from a single vendor. At some stage, it will be necessary for an adopter of the model to make a decision regarding the relative importance of these attributes. It is not always possible to satisfy all of these goals in a particular implementation or use of the A1✓ model.

2.2. Evolution, reconfiguration and legacy systems

A legacy system is one that interacts with another system outside the universe of discourse of the original system specification. That is, if a system has to interact with some system that was not visible or known in the original specification, then a system is a legacy system. All systems will become legacy systems at some point.

To support interoperability with legacy systems, and to provide flexible, robust and configurable distributed systems, implementations supporting the A1✓ model should be dynamic in nature. In the model, reconfiguration and evolution of systems and software is treated as the normal rather than the exceptional case. The concepts of dynamic binding and flexible type description are fundamental to this goal.

Our intention is that implementations supporting the model are capable of interworking with legacy applications. While perpetual backward compatibility with legacy systems is possible, most implementations are expected to sacrifice some backward compatibility in order to achieve progress or meet new challenges.

3. THE "BIG PICTURE"

In order to build a useful distributed system, a real-world problem undergoes a series of transformations to an implemented solution. In the A1✓ model, these transformations are based around two component models:

- a specification model for applications, which defines the fundamental abstractions used for design of distributed applications;
- a distributed infrastructure model, which defines the fundamental services provided by the distributed system.

Based on these two models, there are three major transformations necessary to build a distributed system:

- a transformation of the real-world problem to a specification of an application;

  This transformation is necessary to exclude aspects of the problem that cannot or should not be solved by a computer system. The power and flexibility of the specification model is crucial to this transformation step. This step is similar to requirements specification in software engineering, and will be based on a requirements analysis.
- a transformation of the specification into an application program that is executable over the basic services provided by the infrastructure;

  Transforming a specification into an executable program is application programming. Note that application programming can be assisted by a variety of tools, for example, IDL compilers or application frameworks.
- a transformation of local operating systems and hardware into an environment that supports the infrastructure model.

The concepts and rules of A1✓ model are positioned in terms of these two models and three transformations. Our "Development Model for Distributed Systems" is the combination of the models and transformations and is illustrated in figure 1.

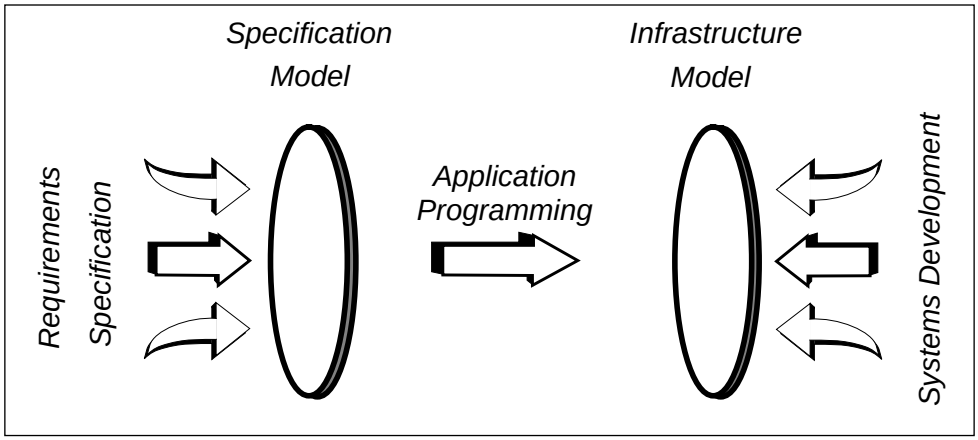


Figure 1: Development Model for Distributed Systems

4. SPECIFICATION MODEL

Distributed applications in the A1✓ model are specified in terms of objects and their behaviour, including interactions. The goal of the specification model is to describe the externally significant behaviour of objects, without regard for how those objects will be physically positioned within the distributed system. That is, the implemented objects could be split across many nodes, or could be combined with others on a single node without violating the specification.

Specifications within this model are intended to describe the primary functionality of the application space, and should not consider the supporting infrastructure. Since the specification model only considers externally significant behaviour, it does not address issues local to objects, for example, concurrency within objects (threads, processes, spawning etc.), but equally does not constrain them. Note that some non-visible behaviour of objects is externally significant, for example, deadlock.

4.1. Objects

An object is an entity that encapsulates state, behaviour and activity. An object has a unique identity, and at any point in time can be associated with a non-empty set of types. An object can evolve over time, changing the set of types satisfied by the object. Objects can only interact by exchanging strongly typed messages. Objects are not passed as parameters to interactions, that is, only identifiers of objects can be passed as parameters (note that this derives from the fact that objects encapsulate activity— activity cannot be transmitted). Objects are autonomous, that is, they are capable of independent action.

Object granularity is the responsibility of the application designer, so the designer can specify the distributed application in terms of individual integer objects if desired, or might equally specify the application in terms of database table objects. The model is object-based—we avoid the use of the term “object-oriented” since inheritance and polymorphism are not explicitly specified.

4.2. Bindings

A binding is a association between a set of objects that allows the objects to interact. Bindings are strongly typed—a binding type defines the roles of objects in a binding and the interaction that can occur between objects fulfilling those roles. In order to provide a general model for distributed applications, we do not prescribe any specific binding types—if desired, it is possible to define binding types specific to each application. The behaviour associated with a binding can be infinite. Bindings are explicitly identified, meaning multiple bindings can exist for any given set of objects.

Roles of a binding are filled by the objects participating in the binding. For example, a binding to support RPCs must have a “client” role and a “server” role. Each role of a binding specifies an interface type that must be satisfied by objects fulfilling that role—objects participating in the binding must therefore instantiate an interface that is a compatible with their role in the binding. Taking our RPC example, the client role specifies that the client object requires a set of operations to be implemented by the server in the binding, whereas the server role specifies that the server object must implement the operations required by the client.

Note that the actual interface types offered by objects for a binding need only be compatible with the roles specified in a binding. In general, compatibility is defined by “substitutability”, that is, the ability of one object to substitute for another object that offers exactly the requested

interface type. For example, a file object offering “open-read-write-close” operations is able to be used for a binding which only requires “open-read-close”.

Although our model permits arbitrary binding semantics, it is expected that a number of common binding types will be used regularly. To simplify the use of such binding types, parameterised, pre-defined types can be used, for example:

- DCE RPC binding
- Unix pipe binding
- ftp binding

Note that to fully describe a binding, these generic types must be parameterised with the interactions and data types (e.g. as defined in DCE IDL) for a specific binding. Higher level binding types can also be defined, capturing the common semantics of several types of binding, for example generic RPC binding. The specification of sub-type relationships between the higher level binding types and more specific binding types provide the basis for determining and supporting interoperability between similar systems.

The fundamental unit of interaction in a binding is a single, strongly typed message, although interactions can be considerably more complex. For example, an RPC interaction consists of a pair of messages between two objects, a request and a response, with the response occurring after the request and transmitted in the opposite direction. For ease of specification, generic, widely used interaction types are defined by the model, including operations (RPC), multicast and strongly-typed flows of data.

4.3. Interfaces

An interface is instantiated to fulfil the role of an object in a particular binding. Interfaces are strongly typed—an interface type describes the possible the structure and semantics for interactions of an object during a binding. In other words, an interface type describes the structure of messages and the object behaviour associated with messages sent and received by that object during a binding. An interface type can also describe constraints on interactions—this might constrain the types of bindings in which the object can participate. By definition, interfaces must exist during a binding and hence must be created either prior to or during the creation of the binding.

In the case of interface creation prior to binding, the type satisfied by the interface is static and not subject to negotiation or transformation during binding. Usually, objects are expected to offer interface types that can be bound, but for maximum flexibility, these types will often define additional interactions not required for all bindings. When a binding is created, an object should instantiate an interface satisfying the role assigned to that object in the binding—this interface need not necessarily implement all of the offered interface type. For example, a file object might offer an interface type describing the operations “open-read-write-close”. A particular binding might only require “open-read-close”, so the interface instantiated for that binding could take advantage of the reduced functionality required by the binding to optimise, for example, concurrency control or caching.

4.4. Types and relationships

A type is a predicate describing a set of entities. In the specification model, types are used specifically to classify and describe objects, bindings, interfaces and relationships. In the specification model, it is assumed that type information is dynamically available—objects can dynamically find, interpret and create types. Interactions between objects must satisfy typing

constraints specified by the participating objects and the binding template. Type information can, however, be hidden to satisfy security or other constraints.

The model also prescribes the maintenance of information about relationships between types, including subtype relationships defining the substitutability of types. An explicit definition of subtyping is not given by the model—since there are many different approaches to subtyping, we simply identify the relation and describe how it can be used to support flexible bindings.

The ability to define general relationships is a key aspect of the model. Relationships are simply associations between entities that satisfy some predicate (i.e. type). As described above, information about relationships between types is maintained, but relationships are also necessary for many other purposes, for example, finding an executable program that offers a particular interface type, or finding the object file from which an object was created.

Methods for description, storage and usage of types and relationships is a area of research being developed within the DSTC. Further detail of this research can be found in [5].

4.5. Fundamental activities

A number of fundamental activities of objects have been identified in the specification model, specifically:

- instantiation and deletion of objects;
- creation and termination of bindings;
- interaction between objects;
- object composition.

These are discussed in more detail in [1]. Of particular interest, however, is the creation of bindings.

Objects can create bindings to other objects. In order to do so, an object must provide a template for the binding, identify the objects that are expected to participate in the binding and their roles in that binding. A binding can only be valid if the identified objects are capable of fulfilling the roles specified for them. Objects participating in a binding can determine that such a binding has been created and have access to an identifier for the binding. Third-party binding is permitted, allowing objects to create bindings without actually participating in those bindings.

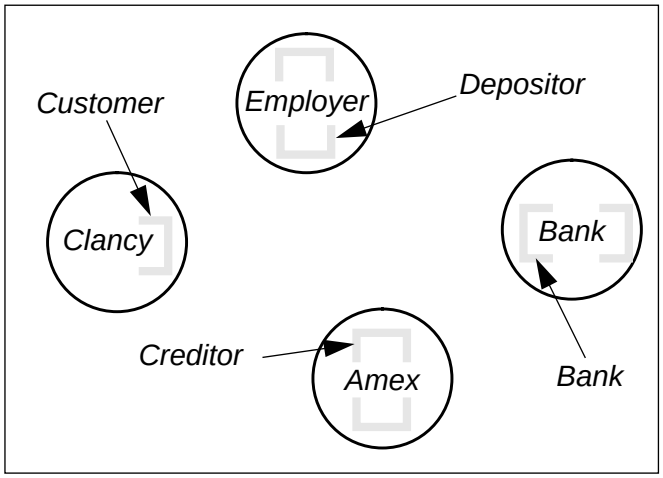
For example, an RPC session between a client and server in a DCE application is a binding. The binding template used to establish this binding is specified by the IDL (interface definition language) file. “Client” and “server” are the roles in this binding. The client is implicitly identified since it initiates the binding. The server is explicitly identified when creating the binding. If the server cannot fulfil the role of server as specified in the IDL, then the binding fails. If the client cannot fulfil the role of client in the binding (e.g. if it does not have stubs for the RPCs defined in the IDL) then the binding fails.

4.6. Illustrating the concepts

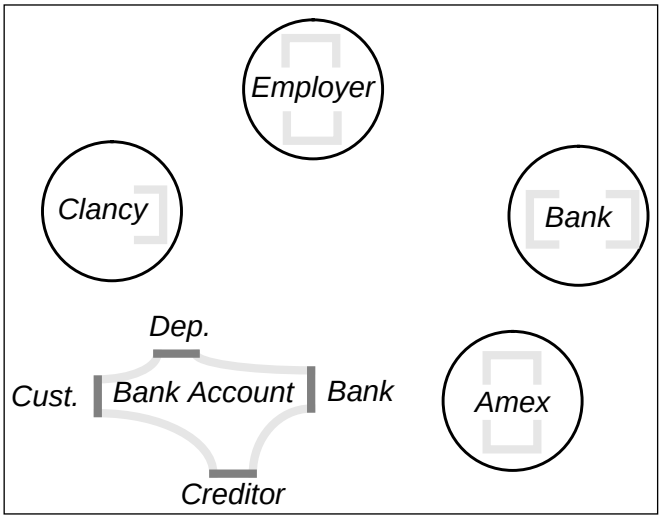
To illustrate the concepts of the specification model, the following diagrams show the three stages associated with establishment of a binding. The illustration shows the creation of a binding between a bank, a customer, an employer, and a credit provider. The binding is intended to support the banking activities of a customer, including automatic withdrawals (e.g. American Express withdrawing a monthly account payment) and automatic payments (e.g. employer pay-

ing salary). We model this activity as a multi-party binding involving all of these objects to capture the dependencies between the interactions.

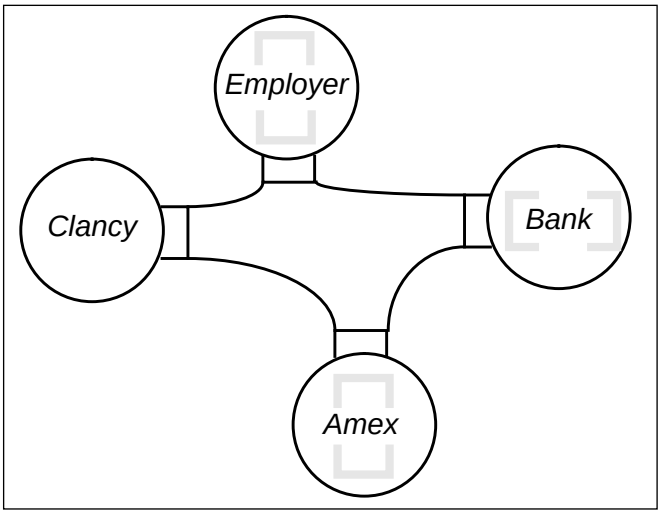
1. Prior to binding, each of the objects assert their ability to support a particular interface type. This is a capability—there is no interface, just an assertion that the object is capable of instantiating such an interface. In this case, we have Clancy asserting the ability to be a bank customer, the employer asserting the ability to be a depositor, and so on.



2. Clancy locates a suitable binding type to support the multi-party banking activity. The bank account binding type has roles (indicated by flat surfaces) for each of the parties. For creation of a binding, the interfaces of the participants must match (subtyping) the roles in the binding type. The binding type specifies the interactions that occur in response to activity at an interface, for example, the pay deposit by the employer might lead to the depositing of the pay at the bank and a notification being sent to Clancy indicating the amount deposited.



3. Once an appropriate binding type has been found, Clancy can initiate the binding. Each participant must agree to their role in the binding, and instantiate an interface that satisfies their role. Note that this need not be exactly the interface type offered—it simply needs to satisfy the associated role. The infrastructure instantiates the binding, connecting each of the interfaces. In this example, Clancy has initiated the binding. In general, however, any object can initiate a binding between any set of objects, including third-party binding (for example, by a trader).



5. INFRASTRUCTURE MODEL

This section describes the high-level design of an infrastructure to support the concepts of the functional specification model. The infrastructure model is intended to be independent of any particular implementation or platform, but aware of the existence of independent nodes and communications between nodes, hence the distribution of objects. The design is expressed as a set of definitions and concepts. The mapping of the specification model concepts to infrastructure model concepts is also described.

The infrastructure provides an abstract machine for execution of object-based program specifications. It enforces the interaction types in the specification, and provides the functionality required to support the abstractions of the specification model.

5.1. Classifying objects

Within the infrastructure model, an object is a unit of distribution (i.e. an identifiable entity that exists on a single node) for distributed applications and services. Note that objects defined in the specification model are not constrained in this manner. Objects are capable of communication through communication networks. The following subsections outline a set of categories for objects that are explicitly identified in the infrastructure model. The definition of the categories is based on the activities or functionality of the objects concerned. More detail for these definitions is given in [1].

5.1.1. Fundamental objects

Fundamental objects are those that are essential to the operation of the infrastructure. That is, they must exist to support the functional specification model. Fundamental objects provide access to the resources offered by nodes and communication networks, for example, communication protocol stacks and operating system kernels.

The activities supported by fundamental objects include instantiation and deletion of entities, that is, objects, relationships, types, interfaces and connections. In addition, naming, encapsulation sufficient for self defence (security), and interaction must be supported by fundamental objects. These activities must be supported by all implementations of the model. Many other activities in distributed systems are important (for example, maintaining security), but are not fundamental since they can be constructed from the fundamental activities. In other words, the fundamental activities are the basis for construction of all other distributed system activities.

5.1.2. Utility objects

Utility objects are objects considered necessary for practical use of the infrastructure. In general, utility objects provide persistent, shared services that will be used by most distributed applications, but are not essential to the functioning of the infrastructure. That is, distributed applications can be built without using utility objects, but it is not generally effective to do so. Some examples of utility objects are:

- a trader object (for service trading)
- a key distribution object (supplying keys for encryption)
- a notification object (providing a notification service for objects)

5.1.3. Auxiliary objects

An auxiliary object supports a particular application or part thereof, but does not directly implement application functionality, and is not essential to the operation of the infrastructure. Auxiliary objects provide programming abstractions or transparencies, and are typically derived

from binding types or from requirements described in the non-functional specification of an application.

Auxiliary objects are obtained either by configuration of “off-the-shelf” components or are application-specific and generated during programming. Some examples of auxiliary objects are:

- RPC stubs which provide marshalling of parameters for RPCs (application specific)
- objects that guard the interactions of an object to provide security (off-the-shelf)

Auxiliary objects differ from utility objects in that they are not shared or persistent, existing only for the lifetime of their application.

5.1.4. Derived objects

A derived object is one that implements part of the application described in a functional specification. The mapping between specification objects and derived objects is not necessarily one-to-one, allowing for the possibility that a specification object is distributed across nodes or that several specification objects are combined into a single implemented object. However, programming tools will support common mappings, for example one-to-one mappings or replicated objects, between specification and derived objects. Derived objects will often include additional functionality required to realise specification-level concepts as infrastructure-level primitives, for example:

- establishing connections to support the bindings described in a functional specification;
- implement interactions in the specification as a set of communication primitives;
- maintaining the consistency of replicas.

5.2. Connections

A connection is a configuration of objects and communication paths that implement a binding (as defined in the functional specification model). A single connection might involve multiple interaction protocols, for example, DCE RPC and FTP.

Connection establishment can involve negotiation between the local infrastructures supporting the interacting objects. A set of connection constraints and parameters compatible with the needs of all objects involved in the interaction are generated. This might include, for example, type matching, data representation or security policy.

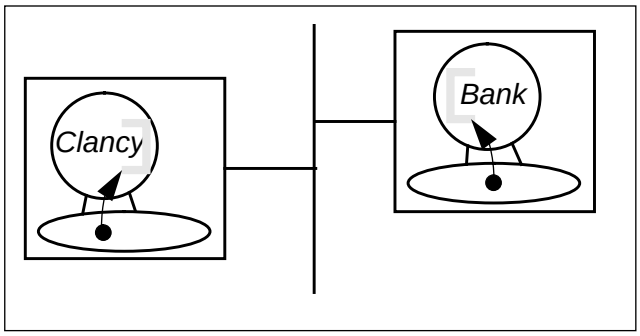
Where the objects participating in an interaction are co-located, some optimisation of the connection process might be possible, but it will still exist as a logical step. The complexity and duration of the connection are a reflection of the interactions possible over the binding. For example, asynchronous interactions are typically very short and might be implemented by packaging the connection establishment and interaction data into a single datagram.

Connections are explicitly terminated. However, connections can also fail due to changes in components of the connection that invalidate the configuration. A connection can be re-established after a failure, and this is the mechanism typically used to cope with, for example, migration or replication of an object during interaction. The termination and re-establishment of a connection does not necessarily terminate the binding implemented by the connection. This allows failure and migration to be transparent in a specification. It also allows the infrastructure to establish and terminate a connection implementing a particular binding on-demand, to optimise resource usage.

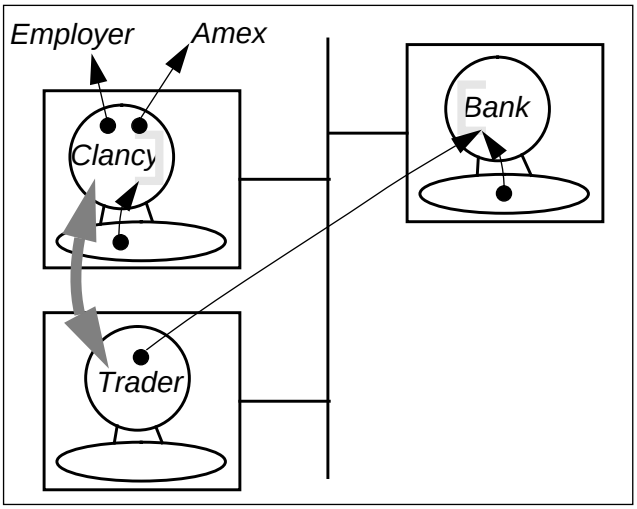
5.3. Illustrating the concepts

In order to illustrate the concepts of the infrastructure model and the relationship to specification concepts, the following figures and text describe the creation of the binding illustrated in 4.6. The major difference between the illustrations is that nodes and networks are now explicitly visible, and must be dealt with.

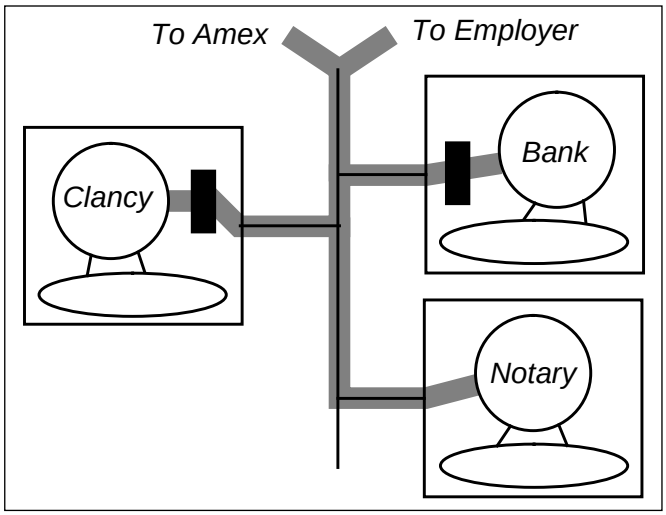
1. The assertion of interface type must be made to the local supporting infrastructure (fundamental objects). This information is needed for establishment of bindings, and is independent of any advertisement of the interface types used for location or resource discovery. We assert that derived objects have pre-established connections with local fundamental objects.



2. Clancy must find each of the other objects to participate in the binding. We assume that Clancy is already aware of the employer and American Express. However, Clancy chooses a bank through interaction with the trader. Note that use of a trader is not prescribed—it is simply used for this illustration (and we do not show the establishment of a connection to the trader).



3. Clancy then asks the infrastructure to create a connection to support a binding type (supplied by Clancy). The connection is created, including auxiliary objects (e.g. security “black boxes” as shown in the figure) and utility objects (e.g. a notary to support non-repudiation) required to support any requirements specified in the binding type. Note that auxiliary objects need not necessarily be co-located with the derived objects and that the connection may require multiple protocols and networks.



6. PROGRAMMING TRANSFORMATION

The aim of programming is to transform a specification into an implementation. Within our model, this means mapping the set of specification objects and bindings onto implemented objects and connections.

Ideally, once the objects, bindings and interactions of an application are specified, the programmer can simply create code for the objects without considering the distribution of those objects in the target system or even considering the fact that the objects are distributed. Realistically, the programmer still needs to consider some aspects of distribution. Transparencies and tools assist (rather than replace) the programmer.

6.1. Transparencies

The infrastructure can be hidden from a programmer through use of transparencies. Transparencies can be achieved through one or more of:

- transformation of specifications (to specifications for programs);
- encapsulation of objects, adding additional functionality;
- using complex bindings to hide, for example, replication.

Transparencies can be implemented through pre-processing of programs, generation of “stubs, or through run-time configuration. The infrastructure can instantiate auxiliary objects to support the transparency requirements of a binding and its participating objects.

7. A1✓ VERSUS RM-ODP

In comparing the A1✓ model to RM-ODP, this section describes how the A1✓ model addresses the deficiencies in RM-ODP identified in the introduction to the paper.

7.1. Viewpoints, models and software development

The A1✓ model introduces two models and three transformations. As pointed out in section 3, the transformations map directly onto software development processes, and the models map onto specification languages and distributed systems infrastructure respectively. The RM-ODP approach of defining five viewpoints is far less clear in this regard.

There is a relatively straightforward mapping between our components models of A1✓ and the viewpoints of RM-ODP. The A1✓ specification model corresponds to the information and computational viewpoints and the infrastructure model corresponds to the engineering viewpoint. Equivalent concepts for the enterprise and technology viewpoints do not feature in the A1✓ model—the development of a distributed application will absorb enterprise and technology modelling into the process (i.e. the transformations).

7.2. Computational language issues

The concept of bindings in A1✓ gives us a simple yet powerful basis to describe both existing and new, complex interaction patterns amongst objects in a distributed application—all interactions are described in terms of strongly-typed messages, with more complex interactions built from this single mechanism. The similar RM-ODP concepts of signals, signal interfaces, binding objects and control interfaces are cumbersome, and are not integrated with the higher-level operational and stream interaction mechanisms and their associated interfaces and types. The plethora of concepts used to describe the three distinct interaction mechanisms complicates the computational language. For legacy systems, the ability to simply and effectively describe the interactions supported by a legacy object is paramount.

The concept of bindings also provides the basis for determining compatibility and subtyping rules. A1✓ allows alternative type rules to co-exist, and this can be captured in a binding type. For example, an ANSAware binding requires ANSAware type rules to be satisfied, whereas DCE binding requires the satisfaction of DCE type rules.

7.3. Engineering and function issues

A1✓ relies on object composition to provide a flexible, generic structuring mechanism in the infrastructure model. The RM-ODP concepts of capsule and cluster provide no additional expressive capability, and the confusion and prescription associated with cluster and capsule managers is avoided.

The clear distinction between fundamental and optional functionality in A1✓ removes the need to prescribe the optional functionality. This makes the A1✓ model minimal yet relatively complete, where RM-ODP describes a number of functions that could well be left for standardisation outside the core model. The optional functionality for A1✓ is being described separately—the DSTC A2 [4], A3 [5] and A4 [6] projects are currently addressing particular aspects of the optional functionality.

8. CONCLUSIONS

The A1✓ model provides a sound basis for the development of distributed applications and infrastructure, and overcomes some specific deficiencies in RM-ODP. These deficiencies have been identified because of the diverse requirements of participants in the DSTC. This paper has described the major concepts of the A1✓ model and discussed the relationship of A1✓ to RM-ODP, in particular, how the A1✓ model addresses the identified deficiencies in RM-ODP.

ACKNOWLEDEMENTS

The authors would like to thank numerous staff of the DSTC, and in particular the Architecture Unit, for lively debate and fruitful discussion. This work is funded in part by the Cooperative Research Centres Program through the department of the Prime Minister and Cabinet of the Commonwealth Government of Australia. This research was also supported by Telecom (Australia) Research Laboratories via the Centre of Expertise in Distributed Information Systems.

BIBLIOGRAPHY

1. A. Berry and K. Raymond (eds), *The DSTC Architecture Model*, DSTC Technical Report, June 1994.
2. ISO/IEC JTC1/SC21, *Draft Recommendation X.902: Basic Reference Model of Open Distributed Processing—Part 2: Descriptive Model* (ISO/IEC DIS 10746-2), February, 1994.
3. ISO/IEC JTC1/SC21, *Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing—Part 3: Prescriptive Model* (ISO/IEC DIS 10746-3), February, 1994.
4. A. Bond and D. Arnold, *Open Distributed Environments—Misadventure or Masterpiece?*, submitted to the International Conference of Open Distributed Processing, Brisbane, Australia, February 1995.
5. W. Brookes and J. Indulska, *A Type Management System for Open Distributed Processing*, TR 285, Department of Computer Science, The University of Queensland, January 1994.
6. A. Beitz and M. Bearman, *An ODP Trading Service for DCE*, Proceedings of the First International Workshop on Services in Distributed and Networked Environments (SDNE), June 1994, Prague, IEEE, p 42-49.