Unity Doesn't Imply Unification *or* Overcoming Heterogeneity Problems in Distributed Software Engineering Environments

M. S. VERRALL
Sema Group plc, Trafalgar House, Richfield Avenue, Reading, Berkshire RG1 8QA, UK

There is much work going on on the separateness of application software in distributed information systems; however, it commonly assumes that the applications will be programmed to the system, rather than the system having to fit itself to the applications. Reverse that assumption and the heterogeneity of the application software becomes a significant problem. This article describes the thesis, mechanisms, and languages for providing solutions to the problem of heterogeneity in distributed information systems for a particular application domain – software engineering.

Received May 1991, revised July 1991

1. INTRODUCTION

Distribution of Information Systems arises from considerations of autonomy, geography, performance and robustness. Autonomy exists because the application software within an Information System may have been created by suppliers with independent objectives but its parts must cooperate to achieve the objective of the Information System. The issues associated with autonomy may be divided into two groups: those associated with heterogeneity, and those associated with separation.

This article is mostly about heterogeneity; it begins with an exposition of the subject of heterogeneity and the statement of the *a posteriori* thesis underlying the Software Bus® work. It then gives a basic description of the Software Bus and elaborates the thesis into three hypotheses. Lastly, it uses these hypotheses to explore a number of interfaces of concern to the Software Bus where autonomy can give rise to problems, and describes the languages and mechanisms in place there.

There is a small glossary of terms at the end.

2. HETEROGENEITY

Heterogeneity of two would-be cooperating pieces of application software may be due to the fact that they have been implemented differently – primarily but not exclusively due to differences in programming language, compiler, and cpu – or that they have differing conceptual models. The current focus of work on Distributed Information Systems, DISs, is the former, but the latter problem lies in wait.

2.1 Heterogeneity of implementation

The usual solution proposed to coping with heterogeneity, i.e. to the problem of getting the differing parts to interact successfully, is 'canonical': a line is drawn and laws are made. This line is the interface to a homogeneous, non-distributed virtual machine. Below the line lie the real, disjoint platforms – whose heterogeneity can be characterised in a simple manner by describing what they provide for execution, communication, and persistence of software and information. Above the line lie the applications in blissful ignorance. The laws that are made are these: (i) the applications must be programmed to the line; (ii) the platforms must be modified (seemingly by extension) to reach up to the line.

® Software Bus is a registered trademark of ESF.

The difficulty with this approach is that it simply does not take into account the fact that there exist both platform and application software whose authors have not the slightest interest in the particular canonical line being espoused. This may be (i) because the software existed before the law, or (ii) the author can see no commercial advantage in programming to the line, or (iii) the author has technical reasons for wishing to program to a different line. Thus the unfortunate would-be builder of a distributed system from an otherwise suitable set of applications and platforms is confronted with the following unpalatable menu of choices: rebuild the applications, write his own, give up – it's all too expensive!

There are many different manifestations of the problem of heterogeneity of implementation and many proposals for solving it of a canonical nature. Furthermore, it seems that when first confronted by this problem, the natural tendency of software engineers is to look to a canonical solution. It would be an interesting problem for psychological investigation as to why this solution is so universally proposed. (Is it that inside every software engineer there is an autocrat struggling to get out?)

2.2 Heterogeneity of concepts

Even if one did adopt the canonical solution to the problem of overcoming differences in implementation amongst application software which wished to cooperate, this would not solve the difficulties caused by a pair of applications which 'don't quite' match.

Categories of near misses include these:

(1) Engineering units. This is when the same information is being measured in different units. For example, personnel management software may wish to allocate people to project control software and be told of people's utilisation. However, people's time may be measured in days by the personnel software, but in hours by the project software.

(2) Information structure. This is when the same item of information is accessed or referred to differently; for example, a person may be identified by national insurance number by the personnel software, but by project name + team number + surname by the project software.

(3) Semantic bridging. This is when a human being can equate two different ideas. An example of a trite but wide span would be to link the reporting function of the project control software – which talks in terms of people and weeks – with the matrix loading function of a chart drawing utility – which talks in terms of rows and columns.

2.3 Coping with heterogeneity

One can arrive at the solution to the system builder's problem by looking at the principal actors in his drama: himself, the application software supplier(s), and the supplier of the integration technology. The supporting cast, including the platform supplier, can be ignored. By recognising that we cannot have all three actors giving orders, and that what the system builder wants from the third actor is the ability to integrate, we arrive at the correct deduction that the integration technology supplier is the one who gets given the orders. After all, 'when you have eliminated the impossible, whatever remains, *however improbable*, must be the truth'.[1] That is, the integration technology does not tell the application software suppliers what they must program to, but the application software suppliers tell the integration technology what they have actually programmed to and the system builder tells the integration technology what the meaning of cooperation is between two parts of an application.

Hence we arrive at the thesis that integration must be '*a posteriori*', that is it must take place by action after the fact. The feasibility of this thesis is demonstrated by the work taking place on the Software Bus, as described in the following sections.

3. THE SOFTWARE BUS

The Eureka Software Factory project, ESF,[2] is a large scale project concerned with creating a class of Software Engineering Environments which support the industrialised production of software. An ESF Software Engineering Environment is a DIS, and is distinguished from traditional data-centric Software Engineering Environments[3,4] by being communications-oriented; it is called a Factory Support Environment, FSE. The communication amongst the application software in the FSE – the software engineering tools and their constituent parts – takes place over a communications channel called the Software Bus. The objective of the Software Bus is the integration of these tools and parts with one another. This objective will be more fully explained by expanding upon those requirements[5] upon the Software Bus pertaining to heterogeneity, showing the position of the Software Bus within the structure of an FSE, and characterising the Software Bus within the DIS world.

3.1 Heterogeneity requirements

From considerations of building and operating an FSE, it can be seen that the Software Bus must

- hide distribution and heterogeneity aspects;
- allow application software to be added without being accounted for *a priori* to an existing DIS;
- allow binding of application software at different times;
- allow exchange of data with as little structural and conceptual loss as possible;
- provide the necessary mechanisms for integration of application software.

From considerations of the quality of support that application software may expect, it can be seen that clients and servers should each

- be able to interact with the other in its own terms;
- not have to know where the other is or that it changes or moves;
- not have to know of the use of multi-casting;
- be able to assume multiple concurrency of the other.

Of course, particular application software need not have been written to comply with such high standards. It is the job of that part of the Software Bus concerned with the installation of applications to upgrade the software being installed, by fitting it with an adaptor, so that it provides – or at least presents the appearance of providing – the requisite quality of support.

As application software may be implemented to interact in many combinations of ways – e.g. procedure call, stream i/o, synchronously, polled or interrupted –

- the Software Bus must cope with a wide repertoire of interaction implementations;
- the arguments of the interactions must allow a wide repertoire of data types.

3.2 Structural view

ESF has chosen to build on the cornerstones of service-oriented building blocks, called Components,* and a communication-oriented architecture.[6] The structural view of an FSE is that of a set of mutually interacting Components, the application software that does the software engineering, joined together by a Software Bus (Fig. 1).

The notion of Component stands at a high level of generalisation. It generalises over code (i) in execution – whether it is a heavyweight or lightweight process – and whether it is taking the role of client, server or both – and its associated persistent data; (ii) as installed and able to execute; (iii) as written, but not necessarily with an executable interface. Components offer each other Services and request the performance of an element of a Service by message passing, not by memory sharing.

This Service Element Request notion also stands at a high level of generalisation. It generalises over all sorts of primitives, and sets thereof, for exchange of control (and thereby, data) embedded in programming languages and operating systems. That is, it generalises over

- data message – the straightforward sending and receiving of data;
- unsynchronised process execution – start, suspend, resume and stop;
- asynchronous process execution – polled or interrupted;

* Throughout this article the term 'Component' is used as ESF jargon for a piece of application software which is a service-oriented building block and which can be attached to the Software Bus.

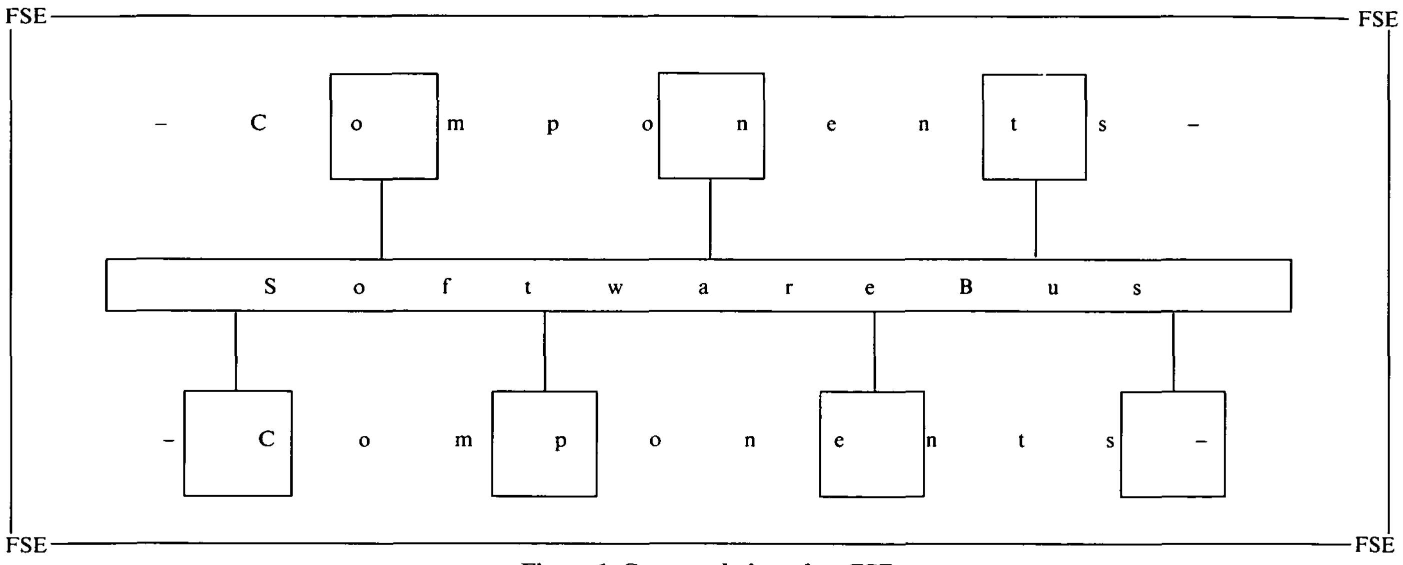

Figure 1. Structural view of an FSE.

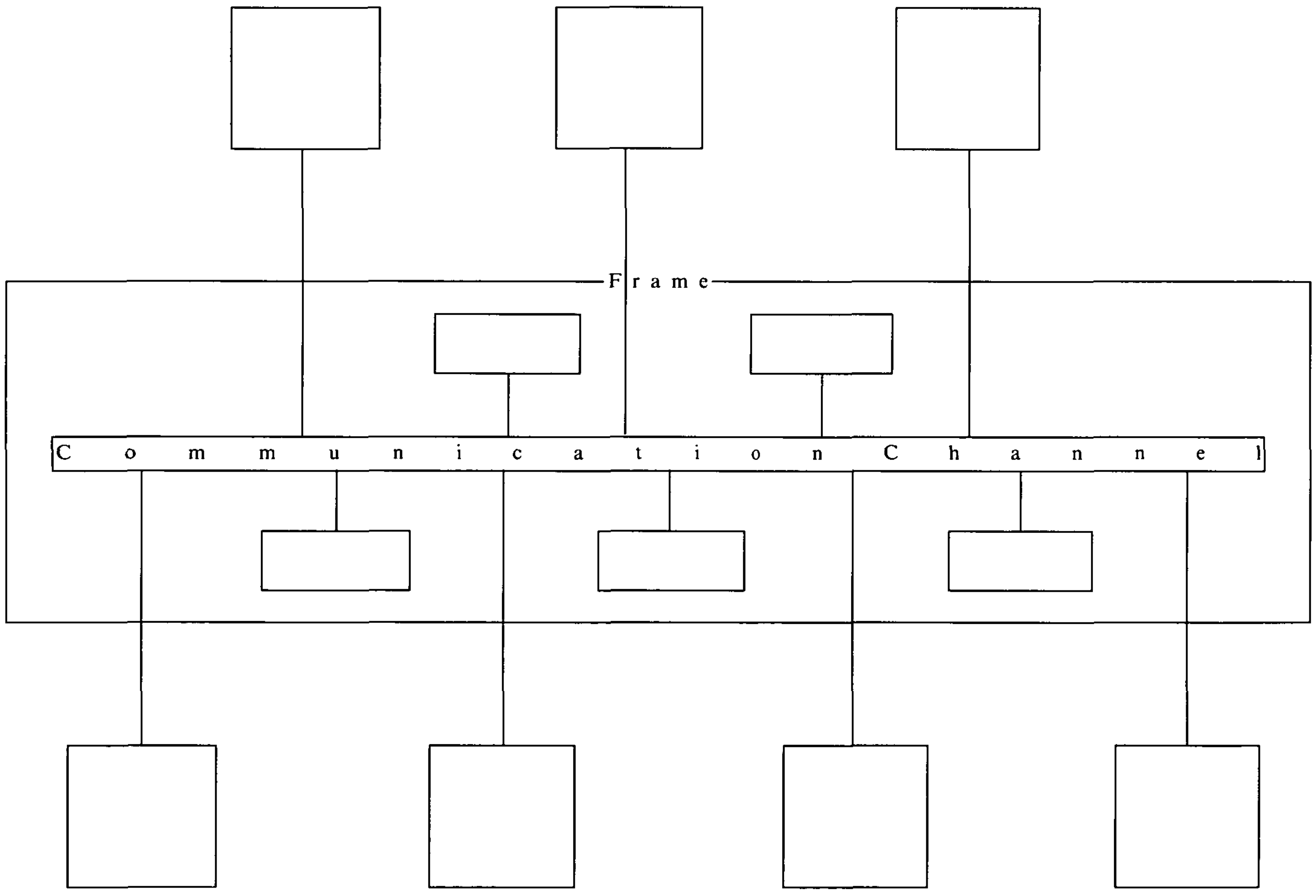

Figure 2. Parts of an FSE.

- synchronous process execution – that is, remote procedure call;
- multiplicity – broadcast, multi-cast and uni-cast.

A Service Element Request may freely overlap

- within itself – there may be a concurrent contraflow of input and output arguments;
- at the client – one Service Element Request may be placed while still placing another (even within one thread);
- at the server – one Service Element Request may be accepted while still accepting another (even within one thread);
- on the Software Bus – with other Service Element Requests.

Service Element Request argument types generalise over the data-types in many programming languages and then abstract further to a repertoire of primitives and constructors useful for software engineers.

The architecture of the Software Bus is itself recursive upon the ESF architecture, in that it is made of a set of Components joined together by a Communication Channel (Fig. 2); the complete aggregation of these Components and the Communication Channel is called the Frame.*

3.3 Characterising the Software Bus

There are a number of ways in which the characteristics of the Software Bus can be understood. It can be characterised in terms of multi-machine operating systems, Open System Interconnection, Open Distributed Processing, and Software Engineering Environment standardisation and development.

3.3.1 Multi-machine operating systems

One may discriminate between the terms 'networking' and 'distributed processing' by saying that networking is interconnection allowing remote access to independent information and processing resources, and that distributed processing combines resources into an integrated structure; thus the Software Bus is concerned with distributed processing. One may distinguish a network operating system from a distributed operating system by virtue of the fact that the former has a very poor level of distribution transparency; as each Component will sit in a virtual machine 'of its own making', it will see exactly as much distributedness as it expects to see (often none); thus the Software Bus can be classified as a distributed operating system, rather than as a network operating system.

If one says that a distributed operating system is an extrapolation of a single-machine operating system, in that it is concerned with the unification of a multiplicity of cpus and devices, either as a distributed kernel concerned with the management of processes and storage (e.g. Mach)[7] or as a conjoiner of single-machine operating systems, then the Software Bus cannot be classified as a distributed operating system. Indeed, it refers elsewhere for intra-system optimisation.

If, however, one regards an operating system as a manager of the resources of a particular (physical or virtual) machine, then the Software Bus – by virtue of the fact that it manages the virtual machine whose resources are Components – can be classified as a distributed operating system. The Software Bus is not an operating system utility such as a distributed file store.[8]

The Software Bus is also concerned with security and reliability: to overcome lack of faith in the truthfulness or robustness of platform (node or connection), client or server. These issues revolve around the interchanges between Components (Service Element Requests), not the items of information managed by Components.

* The difference between the Software Bus and the Frame is the part of the Software Bus which support Components or the FSE at earlier stages of their life cycles, before the FSE is up and running.

3.3.2 Open Systems Interconnection

The part of the Software Bus responsible for the transport of Service Element Requests is the Communication Channel. As a Service Element Request is an abstract thing the Communication Channel could be described as an 'abstract transport layer' in an uneasy attempt to align it with the Open Systems Interconnection reference model.[9] As the Communication Channel can be made of distinct and different physical communication sections joined together, and thereby present the Service Element Request with different representations at different locations, it must in an actual sense cover the Presentation, Session, and Application layers of the Open Systems Interconnection reference model.

3.3.3 Open Distributed Processing

The five Advanced Networked Systems Architecture[10] views of a DIS and the concerns of the Software Bus therein are as follows.

(1) The enterprise view of a DIS explains its place within an organisation. An FSE lies within an organisation for producing software called a Software Factory. The Factory Process Engine[11, 12] is that part of an FSE which enacts a model of the software engineering process linking up the people, activities, software engineering tools and items of information. It is concerned with the enterprise projection; however, the Software Bus is aware of concepts from this view such as person, role, and society for the purpose of identification.

(2) The information view of a DIS is concerned with information and its processing. Within this projection the concerns of the Software Bus are the location and access of the software managing the item of information which is the object of the Service Element Request, it is not itself concerned with the management of items of information.

(3) The computational view of a DIS shows the organisation of the DIS as a set of linked application programs. The concerns of this view – data representations, programming languages, operating system functions, modularity, parallelism, distribution transparency – are the reason for existence of the Software Bus.

(4) The engineering view of a DIS describes the processing, memory and communications functions that support the application programs of the computational projection.

(5) The technology view of a DIS describes the physical hardware and software parts that make up a DIS.

The important fact here is that the Software Bus exists within and extends into other people's engineering and technology views. The Software Bus is not built on top of other people's systems, but projected into their systems' universes of discourse – which is a different way of thinking about integrating a DIS.

3.3.4 Software Engineering Environment standardisation and development

The ECMA[13] reference model for Software Engineering Environments divides the facilities that a Software Engineering Environment framework should provide for the software engineering tools into a number of services* including data repository, data integration, task management, and messages. The message service should provide a standard communication service which can be used by a tool or service to communicate with another tool or service. The Software Bus can be placed here in the ECMA reference model. The key architectural features currently being worked on in Software Engineering Environments include process modelling, object frameworks, and message passing; these all contribute to the integration of a Software Engineering Environment.[14] Sometimes messages are passed directly from tool to tool, but more prevalent is the use of dispatchers and routers to pass the messages; the Software Bus is an example of message passing by dispatcher and router.

The Software Bus is not an object management system: it neither unifies Components into one global object management system nor supplies object management systems as platforms for Components nor concerns itself with the objects inside Components.

The nature of the interacting pieces of software that the Software Bus is concerned with is large scale. Typically they are computing engines, object management systems, user interaction management systems, not the small-scale items of information – though, of course, the Software Bus is aware of the existence of these items. The things that the Software Bus sets out to locate, associate, bind, compose and configuration-manage are Components, not items.

4. HYPOTHESES

To see how the thesis that integration must be *a posteriori* manifests itself at the interfaces of the ESF architecture and in the mechanism of the Software Bus, it is first necessary to elaborate it into a number of hypotheses.

In order to integrate after the fact it is necessary to be able to say what the fact is. That is, we must have languages to describe what actually holds in all zones of integration, rather than rules which prescribe what ought to hold. Thus we have our first hypothesis:

Be descriptive, not prescriptive.

As we do not have a uniform situation with respect to conformance to a set of rules, we must decompose what we talk about in the actual situation into different realms – such that what holds in each realm can be described by a uniform language for that realm. By considering the structural view we see that we can properly and naturally separate the realms of Component and Service. A Service is not just an interface to a particular Component, but has an independent conceptual existence. Indeed, it is possible to have dual perspectives on a DIS, either as a set of Components joined by their requiring and offering of Services or as a set of Services joined by their realisation by Components. Thus we have our second hypothesis:

Separate the concepts of Component and Service.

Reviewing the discussion of the aspects of heterogeneity earlier we can see that a Service can be partitioned into the realms of concept and implementation. In the conceptual realm lie the independently existing Services abstracting over all and any implementations thereof; in the implementation realm lie the representations of the Services which link to the realm of the Components. Thus we have our third hypothesis:

Distinguish between Service abstractions and representations.

Compared with existing Interface Definition Notations,[15,16] this makes explicit the separation of abstraction and representation.

5. LANGUAGES AND MECHANISMS

The linkage of the FSE, both in a conceptual and operational sense, to the real world in which it is embedded – the Software Factory – is performed by the Factory Process Engine. Within the FSE the Factory Process Engine seeks to activate software tools; it is to this activation that the Software Bus must link. To see what the mechanisms are that the Software Bus puts in place to enable Components to be created, joined together and linked to the Factory Process Engine, it is necessary to develop the following vocabulary.

The term Component must be specialised to:

Body. The code and persistent data within a Component Executant which actually does the work of the application.

Composite. A Component whose internal structure conforms to the ESF paradigms, i.e. it is made up of Components.

Executant. An executing piece of software, including its persistent data, which is supplying and using Services within the FSE.

Installant. A piece of software within the FSE capable of giving birth to a Component Executant.

Skeleton. That part of the Component Body which corresponds to the interface descriptions of the Services provided and required by this Component.

Tool. A Composite Component identified with a software engineering tool of the Factory Process Engine.

There are three types of Description associated with Service:

Abstraction Description. A description of a Service independent of the software form of its provision or request.

Interaction Description. A description of how a set of Services described by different near miss (§2.2) Abstraction Descriptions can interact.

Representation Description. A description of how a Service described by an Abstraction Description is provided or required in software.

There are two parts of the Software Bus of particular interest:

Abstraction Converter. A piece of software which converts Service Element Requests from those of one Abstraction Description to those of another, according to an Interaction Description.

Representation Converter. A piece of software which converts Service Element Requests, from the rep-

* The ECMA reference model's usage of the word 'service' differs from that of the Software Bus.

resentation given by one Representation Description to that given by another Representation Description.

These are explained and exemplified in the following sections:

§5.1 Abstraction Description;
§5.2 Representation Description, and its relationship to Component Skeleton and Representation Converter;
§5.3 Interaction Description, and its relationship to Abstraction Converter;
§5.4 Component Type, and the relationships between Component Body, Installant and Executant;
§5.5 Composite Component;
§5.6 Tool.

5.1 Service Abstraction Description

A Service Abstraction is described in a fashion that generalises over all the ways in which a Service could be requested in programming languages, as discussed in §3.2.

At the abstract level the Services are described within the Abstract Data Type paradigm and thus each is encapsulated in a set of operations – the Service Elements – through which they are accessed by the Service Element Requests. They are expressed in the most abstract representation as a tuple of <name, reciprocating control exchange, arguments given out, arguments given back> and thus appear like an ordinary local procedure call. The arguments can be values, but they cannot be items of information (i.e. the actual data entities holding the values) or Components. The values are of data types defined or constructed in the language for describing Services in the abstract; if a value is interpreted by one or more Components as a reference, this is the business of the Components and their platforms, not of the Software Bus.

As there is inheritance in the Service Abstraction Description Language, Services can be built upon each other, rather than from the ground up, by allowing the Abstractions to inherit one from another. However, the verification of subtyping between Services is more complex than that for data-type definitions in third generation languages; amongst other things the verification rules consider argument direction, type and default value existence; thus they are similar to, but more extensive than, the rules for type conformance in the Emerald language.[17]

5.1.1 Example Service Abstraction Description

The following example (Fig. 3) shows selected fragments of a Service Abstraction Description, which has been chosen from the domain of project and team management. The example is given in one particular concrete syntax for a Service Abstraction Description Language in which keywords are in upper case.

In the example domain, a project has a name and is made up of a number of teams, and a team has a name and a number of staff members who are identified by their surnames.

The fragments show these descriptions:

- the declarations of data types for the name of a person, project and team. The name of a team can be seen to be built upon the name of a project.
- a Class called '**`Team1`**' which expresses a team and has only one method, which is a query. A Class is two things: it is a data type of the Service Abstraction Description Language (it is an encapsulated data type), and it is a Service. As it is an encapsulated data type, each method contains an argument of type of that Class; in this case the argument **`On`**. As it is a Service it may be offered and required by Components. The method **`WhatStaff`** returns the staff on the team as a bag – as a surname can occur more than once, this is the correct level of abstraction.
- a Class called '**`Team`**'. This inherits from '**`Team1`**', as shown by the braces, and adds methods to create a team and manipulate its staffing.
- two Services for **`TeamLeading`**. These Services express the operations of some functionality, rather than the encapsulation of data. The duplication of Services and difference in return arguments are used in the development of this example in §5.3. The return argument in the '**`B`**' Service express the fact that the data structure being passed in the argument is a tree – a level of abstraction that a software engineer often wants to express, but which cannot be expressed in common programming languages.
- similarly, a Class, '**`Project`**' and a Service '**`ProjectManagement`**'.
- a Service which expresses the functionality which can take charge of managing a project. As the example develops in §5.5, the actor ultimately behind this will be seen to be a human being rather than software.

The sub-division of Services into those which express encapsulated data and those which express functionality can be compared to the data type and functional modules of some principles of software architecture,[18] and more generally to the classification of a vast number of software engineering methods according to whether they place greater emphasis on information or process.

5.2 Service Representation Description

A Service Representation is a set of rules for mapping the requests associated with one or more Service Abstractions onto the virtual machine that the Component Body operates on, this consists of some or all of the following parts

- the programming language's paradigm;
- the language's interpreter or run-time system;
- intra-process schedulers;
- operating system process execution control;
- levels built on top of these.

Also, the programming language in which the Component Body is written may have representation control primitives in it, e.g. the representation clause in Ada,[19] or may be subject to some manually enforced coding rules, e.g. 'even in Fortran always encapsulate'; the descriptive power of Service Representations is being developed to cope with these.

There are two uses for Service Representations:

(1) Component Skeleton Generation. To assist builders of new Components, a Service Representation Description may be combined with a Service Abstraction Description to generate a Component Skeleton.

(2) Representation Converter Generation. For both new and existing Components, it will be necessary to

```
TYPE      Letter_String;
            -- the definition of Letter_String is not shown

TYPE      Surname IS
          LIST OF LetterString; -- to allow hyphenated names

TYPE      ProjectName IS
          RECORD (* various fields *) RECORD-END;

TYPE      TeamName IS
          RECORD {ProjectName}
          Team: LetterString; -- extends ProjectName by one field
          RECORD-END;

CLASS     Team1 IS

    METHOD WhatStaff [On:Team1]
    RETURNS [Staff: BAG OF Surname] -- duplicate surnames possible

CLASS     Team IS {Team1}

    METHOD CreateTeam [Called:TeamName]
    RETURNS [New: Team]

    METHOD AddStaff [Called: Surname; To:Team]
    RETURNS []

    METHOD RemoveStaff [Called: Surname; From:Team]
    RETURNS []

SERVICE   TeamLeadingA IS

    SERVICE-ELEMENT WhatTeams []
    RETURNS [Teams: LIST OF Team]

SERVICE   TeamLeadingB IS

    SERVICE-ELEMENT WhatTeams []
    RETURNS [Teams: TREE OF Team]

CLASS     Project IS
    METHOD AddTeam [Called:TeamName; To:Project]
    RETURNS [New: Team]

    METHOD ShowTeams [In: Project]
    RETURNS [Teams: SET OF Team]

    METHOD MoveStaff [Called: Surname; From, To:Team; Within:Project]
    RETURNS []

SERVICE   ProjectManagement IS

    SERVICE-ELEMENT ManageProject
           [Called: ProjectName; WhichIs: {New, Existing}]
    RETURNS []

SERVICE   ProjectManager IS

    SERVICE-ELEMENT Manipulate [Project: Project; Called: ProjectName]
    RETURNS []
```

Figure 3. Service Abstraction Description.

generate a Representation Converter, which becomes part of the Component Installant, and thus part of the Component Executants enlivened from it, so that the Component Executants can be plugged in to the Communication Channel. Generation takes a Service Abstraction Description and makes a difference comparison of two Service Representation Descriptions. The transfer syntax that a particular Representation Converter converts to can be fixed when it is generated or can be negotiated later on in the Component's life.

5.2.1 Component Skeleton

A Component which offers or requires a Service has to express the ability to do so in its programming language. The statements which allow Service Element Requests to be accepted or placed, together with their supporting declarations, constitute the Component Skeleton.

To represent a Service, whether it expresses functionality or encapsulated data, it is necessary to represent the set of Service Element Requests that can be placed or accepted. The representation of a request is concerned with representing two things: the act of requesting itself and the arguments of a request.

The request itself is represented as a (set of) control exchange primitive(s) within the programming paradigm of the Component Body; this may be anything from a simple local procedure call to an operating system directive for inter-process control or communication. The point is that the Software Bus is undemanding here and will allow the programmer the choice of coding.

The argument types are at a higher level of abstraction than the data types of many programming languages and thus must be suitably related to them.

An argument type of Class, which means a reference to an instance of the encapsulated type, can be represented as an object identifier in object-oriented languages, but in procedural languages there is no such concept. In this case such an argument is represented by a 'handle' variable which is opaque to the programmer. This is acceptable as the Component Body will only ever pass this value between Service Elements and not try to access the instance of the Class itself – this is in accord with the fact that the Software Bus does not move items of information around the FSE.

An argument whose type is a data type in the Service Abstraction Description Language may or may not be able to be represented directly as a data type of the programming language. Consider the data types in the example in §5.1.1 and some common programming languages; `LIST` is only in Lisp, `SET` only in Pascal, `BAG` and `TREE` in none, and Fortran cannot even manage `RECORD`. When this arises the representation of such 'non-native' data types within the programming language can be

- a collection of data types native to the programming language, together with a collection of consistency rules which must hold;
- a set of routines encapsulating some native data types (which thereby enforce the consistency rules).

The routines allow access to the non-native data types, usually in a navigational fashion in order to fit with the programming language's paradigm. The routines can be provided either by the Software Bus to the programmer or vice versa. The early prototypes of the Software Bus are supporting representation of non-native data types by its own encapsulating routines, subsequently it will permit the other representations.

Finally, if the Component Skeleton is to be generated, it must be delivered in a form suitable for use by the programmer; the files in which it is delivered will be destined for inclusion in the Component Body or for separate compilation, and they may contain statements which look like programming language statements but which require macro processing. All this varies with the language and its programming environment.

For example, the Class **`Team`** can appear quite naturally as the Ada package in Fig. 4.

This shows the automatic construction of names from both named and unnamed entities in the Service Abstraction Description; however, if the programmer wished to use different names, it is only necessary to say so in the Service Representation Description. It also shows the use of a handle, the type TeamsAndProjects_Team, to identify different instances of the Class.

In many languages, such as C, there is no concept akin to packaging, so a Class is represented as a set of external functions and the handle is a record data type – the programmer has to be trusted to leave variables of this type alone.

The next example deals with the problem of non-native data types and shows how the ability to manipulate a `LIST` argument is supplied for and used in a Component written in C. An include file (Fig. 5) declares the encapsulating operations on `LIST`.

Then a fragment of the Component Body to count the number of teams in a project could look like Fig. 6.

The fact that the fragment has not been coded as part of a distributed application, and thus does not expect a distribution error status to be returned by TeamLeading-

```
package TeamsAndProjects_Team_CLASS is

   type TeamsAndProjects_Team is limited private;

   procedure TeamsAndProjects_Team1_WhatStaff
      (On: in TeamsAndProjects_Team;
       Staff : out TeamsAndProjects_I_t0;);

   procedure TeamsAndProjects_Team_CreateTeam
      (Called: in TeamsAndProjects_TeamName;
      New : out TeamsAndProjects_Team;);

   -- etc.

 end TeamsAndProjects_Team_CLASS ;
```

Figure 4. A Class as an Ada package.

```
typedef struct {unsigned opaque[9]} List;
typedef void *LstIter;
typedef struct { /* etc. */ } ListStatus;

extern LstIter Create_Cursor();
extern void Drop_Cursor();
extern int End_of_List();
extern void Next_in_list();
```

Figure 5. Supply of non-native types in C.

```
List *teams;
LstIter cursor;
ListStatus status;

TeamLeadingA_WhatTeams(teams);
cursor = Create_Cursor(teams);
count = 0;
for (; !End_of_List(cursor);
Next_in_List(cursor,&status))
    {count++;}
Drop_Cursor(cursor);
```

Figure 6. Use of non-native types in C.

A_WhatTeams, means that distribution fault transparency is perforce provided by the Representation Converter.

At the conclusion of this section the reader may be led to remark that the programming language fragments presented are all very typical of the languages in question and do not present great novelty to the programmer of the Component Body. This is because that is precisely the desired effect, which is necessary to permit *a posteriori* integration of (non-distributed) application code without generation of a Component Skeleton or modification of the Component Body (which already exists).

5.3 Service Interaction Description

A set of Services can be associated together and the role that each Service plays in such an association, e.g. 'client', 'server', 'side effect', be given – thus defining the interacting pairs within the association and the way in which they interact. It is possible that one Service can occupy more than one role.

Though there may be a difference between the paired Services at the abstract level, it may be possible for the human being to cross-reference the semantics and thus close the gap by declaring truths that hold in comparing two Services. The predicates expressing these truths constitute the Service Interaction Description.

For example, there may be a Component which requires a Service for team leading and another which offers it, and the opportunity to couple them together within an FSE arises. However, closer examination shows that the client requires the Service described by **TeamLeadingA** but the server provides **TeamLeadingB** and that these are not the same – the category of mismatch is that called an information structure near miss in §2.2. Thus, a Service Interaction Description is written which states that

- **TeamLeadingA** is tantamount to **TeamLeadingB**
- **TeamLeadingA WhatTeams** corresponds to **TeamLeadingB WhatTeams**
- the application of specific list and tree navigating operations to the differing arguments of the different **WhatTeams** in a specific order will yield the same **Team.** (The actual predicates formed from the navigational operations are derived from the ordering and structuring rules applicable to these arguments).

This Service Interaction Description is used in conjunction with the Service Abstraction Descriptions of the two Services that it relates, **TeamLeadingA** and **TeamLeadingB**, to generate the Abstraction Converter that will permit the Component Executants on either side of the correlated Services to cooperate. The Abstraction Converter is part of the Frame, in that it is itself a Component.

5.4 Component Type

Having created a Component Body, either directly or from a Component Skeleton, and used the Service Descriptions to generate the Representation Converter, the Component Builder assembles the Component Body and Representation Converter to build the Component Installant. The Component Installant is the manifestation of the Component which can be installed in an FSE; but, before doing so, the Component Type Description must be deposited in the FSE. A Component Type Description describes the Services provided and required by a Component and their representation, by reference to Service Abstraction and Representation Descriptions.

Take as an example a Component which is a project management engine. It provides the mechanisms to perform project management, relies on the provision of team management mechanisms, assumes an interface to a project manager (human being) to instruct it, and is written in Ada. Its Component Type Description would state that it

- `provides Service` **`ProjectManagement`**
- `requires Service` **`ProjectManager`**
- `provides Class` **`Project`**
- `requires Service` **`TeamLeadingA`**
- `requires Class` **`Team`**
- `represents these according to a Service Representation Description which describes a straightforward Ada package representation`

Once the Component Installant has been installed, one or more Component Executants can be enlivened from it ready for execution, that is the placing and accepting of Service Element Requests, in the FSE.

5.5 Composite Component

Typically, a Component will not be complete in itself and will be assembled together with other Components to make something more useful as a Composite Component. A Composite Component Executant is made up of Component Executants (which may themselves be Composite), rather than programming language entities. A Composite Component has a type given by its Description. When a Composite Component Executant is enlivened the Composite Component Description is consulted and the appropriate (Composite) Component Executants enlivened and bound together. A Composite Component Description is written, not by a Component Builder, but by an FSE Builder, thus the job of

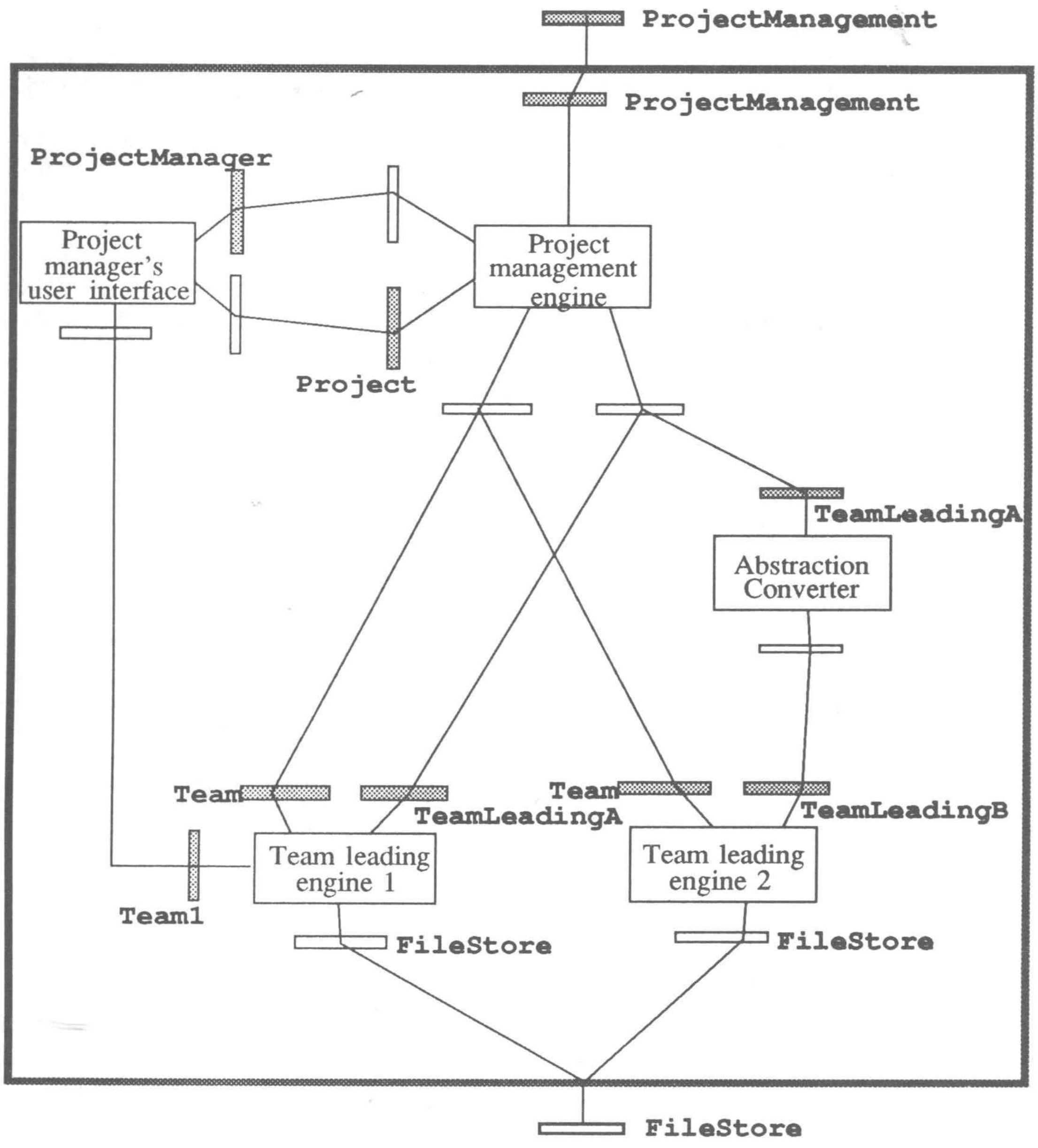

Figure 7. Composite Component for Project Management.

application programmer can be seen to be split into two roles fulfilled by two different people.

An example Composite Component Description is illustrated in Fig. 7 using a graphical concrete syntax for the Composite Component Description Language. In this syntax, the Composite Component being described is shown as a bold rectangle, the Components of which it is built are shown as plain rectangles, the Services are shown as bars – filled for offered, hollow for required – and the bindings are shown as lines. As bound Services are the same, only offered Services are identified, in general, to avoid cluttering the diagram.

The example Composite Component provides project management, has a user interface to a project manager and needs only a file store. It is made up of

- the project management engine discussed in §5.4;
- two team leading engines:
  these offer slightly different team leading Services, and one of them allows inquiry into the staffing of its teams by the **Team1** Class;
- a project manager's user interface:
  through this the project manager can manipulate projects and look at the staffing of teams (because of the composition and structure of the Composite Component this operation is only open to him for certain teams);
- the Abstraction Converter discussed in §5.3.

It illustrates:

(1) Topology. The Composite Component is a directed graph.

(2) Identicalness of paired Services, both of the bound Services within the Composite Component and of the Services of the Composite Component corresponding to Services of Components of which it is made.

(3) Multiplicity of bindings. A client may be bound to multiple servers for the same Service.

5.6 Tool

A Tool is a Composite Component which is made known to the Factory Process Engine as a Tool in its conceptual world. When the Factory Process Engine starts a Tool it commands the Software Bus, which as a result enlivens a Composite Component and sends it the appropriate Service Element Request to get it going.

A Tool has a type given by its Tool Type Description, which relates it to a Composite Component and says how to transform control between Tool and Composite Component.

Continuing the example in §5.5, a further Composite Component could be assembled from the Composite Component in §5.5 and a file store Component and could then be equated to a Tool in a Tool Description. Then the command by the Factory Process Engine to start this particular tool would ultimately be transformed to a Service Element Request **ManageProject** to an enlivened project management engine Component.

Of course, the Factory Process Engine is itself a Component within the FSE and so the command to start a tool is itself a Service Element Request and the Software Bus paradigm is maintained.

6. CONCLUSION

The above languages and mechanisms describe the Software Bus, which has been elaborated from the basic ESF structural view by the application of some hypotheses arising from the application of the thesis of *a posteriori* integration to this view. In order to validate this a number of parts have been prototyped, particularly the Service Abstraction Description language; the validation is currently being extended by construction of a long, shallow and narrow Software Bus implementation.

This work helps advance thinking on Distributed Information Systems from autocratic to democratic. Its principles should be true for any DIS, as they have been derived in a general way. The limitations will arise from the fact that some details of the languages of the Software Bus are motivated by the need to support software engineering applications and may not therefore be sufficiently wide ranging to cover all DIS applications.

There is a rich seam of future work to be exploited; the Software Bus can be extended in length (coverage of DIS issues), depth (richness of languages and mechanisms), and width (of spectrum of DIS applications).

7. GLOSSARY

- Component – a piece of application software which is a service oriented building block and which can be attached to the Software Bus.
- DIS, Distributed Information System.
- ESF[2], Eureka Software Factory – a project to develop Software Engineering Environments for industrialised software production.
- Factory Process Engine[11, 12] – that part of an FSE which enacts a model of the software engineering process.
- FSE, Factory Support Environment – an ESF Software Engineering Environment.
- Service – a set of operations within the FSE whose provision or requirement is discussed as a set.
- Service Element Request – a request for the performance of an element of a Service; it is placed by one Component and accepted by another.

Acknowledgements

The author gratefully acknowledges the participation of the other members of the Software Bus team in the development of the ideas and software of the Software Bus and the suggestions of the referee for improving the presentation of this article.

REFERENCES

1. A. Conan Doyle, *The Sign of Four*. Spencer Blackett, London (1890).
2. C. Fernstrom and L. Ohlsson, ESF – An approach to Industrial Software Production. In *Software Engineering Environments: Research and Practice*, Proceedings of 4th Conference on Software Engineering Environments (Durham, April 1989), edited K. H. Bennet, pp. 17–28. Ellis Horwood, Chichester (1989).
3. M. S. Verrall, SFINX Project – The Componentry Approach to SEE Building. In *Software Engineering En-*

vironments: Research and Practice, Proceedings of 4th Conference on Software Engineering Environments (Durham, April 1989), edited K. H. Bennet, pp. 173–183. Ellis Horwood, Chichester (1989).
4. M. S. Verrall, Tool interaction and integration in a software engineering environment. In *Proceedings of 1st International Conference on Software Development Environments and Factories* (Berlin, May 1989), edited N. Madhavji *et al.*, pp. 229–239. Pitman, London (1990).
5. M. S. Verrall, The Software Bus – its objective: the mutual integration of distributed software engineering tools. In *Proceedings of 5th Conference on Software Engineering Environments* (Aberystwyth, March 1991), edited F. Long. Ellis Horwood, Chichester (1991).
6. *ESF Technical Reference Guide, Version 1.1*. EUREKA Software Factory, Berlin (1989).
7. R. F. Rashid, Threads of a New System. *UNIX Review* **4** (1986).
8. M. Satyanarayanan, Scalable, Secure, and Highly Available Distributed File Access. *Computer* **23** (5), (1990).
9. *ISO 7498–1984 Information processing system – Open Systems Interconnection – Basic Reference Model*. International Organization for Standardization, Geneva (1984).
10. *ANSA Reference manual, Release 01.00*. Architecture Projects Management Ltd, Cambridge (1989).
11. L. Hubert and G. Perdreau, Software Factory: Using Process Modelling for Integration Purposes. In *Systems Integration '90*, Proceedings of 1st International Conference on Systems Integration (Morristown, April 1990), edited P. A. Ng *et al.*, pp. 14–25. IEEE Computer Society Press, Los Alamitos (1990).
12. V. Gruhn, Analysis of Software Process Models in the Software Process Management MELMAC. In *Proceedings of 5th Conference on Software Engineering Environments* (Aberystwyth, March 1991), edited F. Long. Ellis Horwood, Chichester (1991).
13. *A Reference Model for Frameworks of Computer-Assisted Software Engineering Environments, ECMA TR/55*. European Computer Manufacturers Association, Geneva (1990).
14. L. Osterweil, Software Development Environment Research Projects in the United States. In *Proceedings of 5th Conference on Software Engineering Environments* (Aberystwyth, March 1991), edited F. Long. Ellis Horwood, Chichester (1991).
15. *Network Programming*. Sun Microsystems, Mountain View (1987).
16. Computational model. *ANSA Reference Manual, Release 01.00*, section X.7. Architecture Projects Management Ltd, Cambridge (1989).
17. R. K. Raj *et al.*, Emerald: A General-Purpose Programming Language. *Software – Practice & Experience* **21** (1) (1991).
18. M. Nagel, Modelling of Software Architectures: Importance, Notions, Experiences. In *Software Development Environments and CASE Technology*, Proceedings of the European Symposium on Software Development Environments and CASE Technology (Königswinter June 1991), edited A. Endres and H. Weber, pp. 211–232. Springer Verlag, Berlin (1991).
19. *Reference Manual for the Ada Programming Language, ANSI/MIL-STD 1815A*. Ada Joint Program Office, Washington (1983).

Book Review

T. J. Mayes (editor)
Abstracts in Human Computer Interaction, vol. 1, issues 1 and 2.
Ergosytes Associates. ISSN 1042-0193. Price US$316 in the UK.

This is a quarterly, international journal providing abstracts of selected English-language literature on the human factors of computer systems. Each issue promises to present between 500 and 600 abstracts of relevant material selected according to two criteria: serving the needs of the entire HCI community (a motley crew of disciplines from engineering design to academic psychology) and satisfying the standards of the editorial board, i.e. the cited article should be of refereed journal quality. That there is a need for such an abstracting journal cannot be doubted, given the editor's remarks in issue 1 to the effect that 75% of the HCI literature will have to be rejected if each issue is to be kept to its intended size.

Issues are divided into three sections. The first, a subject guide, groups material according to content area, for example formal methods, adaptive interfaces, etc., thus enabling readers to ignore material of no interest and search through only relevant themes. The details here are of the form: article title, author, publication type (e.g. book, conference paper) and abstract number, all organised alphabetically within sections according to first author's name.

The second section contains the abstract records. These are organised alphabetically with respect to author(s) and are of the form: 'i.d.' number, author(s), affiliation, publication type (right-justified in a left-hand column), with title, source and abstract to the left. Good typography provides a neat presentation style throughout. Readers are advised at the start that some of the abstracts were provided by the editors not the authors. Naturally, this section forms the bulk of the issue. The third (and shortest) section is a straight alphabetical listing of all authors in the issue.

The real test of a publication such as this is the extent to which you use it. It has sat on my desk for four weeks now and strikes me as the sort of tool to which I would gladly give increasing shelf space. Not only does it provide instant access to this rapidly expanding literature but it lets you know what sort of material is increasingly appearing and gives me, at least, a lazy way of storing precise references to material.

Only minor criticisms apply. One is that it takes a few minutes to get used to the layout and organisation of the text. However, once understood this should not be a problem, but I witnessed two colleagues having difficulties locating material with the review issues I showed them. Another is that referencing and then tracing material according to a three-digit code is no match for page numbers. I understand the production difficulties of page numbering, but user preference is an important human factor!

Certain references appear to be incorrect (or at least non-standardised) with respect to publication type. Thus we get details of some papers from the first UK Hypertext Conference in 1988 referred to as book chapters from 1989 (the published proceedings), while another paper at the same conference is described as being a conference paper from 1988. Also the subject classification system is bound to confuse people. Would you expect to find books and papers on 'computer addiction' and the so-called 'scenario methodology' together in a section titled 'User characteristics and Models'? Does hypertext warrant a section of its own or not? Unfortunately no classification will satisfy everyone. Finally, why start at 1988? I know the field changes rapidly, but access to details of material as far back as 1980 is not uncommon in my work, and reading these issues made me wish this journal had existed 10 years ago.

In conclusion, this is a timely and useful journal for the HCI worker and the editors are to be congratulated on their efforts. I'd love it on my desk but at US$300 or so to subscribe I will not be purchasing it myself, and if I have to go to a library every time I want to check something, I fear it will lose much of its value to me.

ANDREW DILLON
Loughborough