Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Prioritized Service Architecture: Refinement and Visual Design

Ian R. EAST

Dept. for Computing, Oxford Brookes University, Oxford OX33 1HX, England

ireast@brookes.ac.uk

Abstract. Concurrent/reactive systems can be designed free of deadlock using prioritized service architecture (PSA), subject to simple, statically verified, design rules. The Honeysuckle Design Language (HDL) enables such *service-oriented* design to be expressed purely in terms of communication, while affording a process-oriented implementation, using the Honeysuckle Programming Language (HPL). A number of enhancements to the service model for system abstraction are described, along with their utility. Finally, a new graphical counterpart to HDL (HVDL) is introduced that incorporates all these enhancements, and which facilitates interactive stepwise refinement.

Keywords. client-server protocol, deadlock-freedom, programming language, correctness-by-design, visual programming, visual design.

Introduction

Background and Motivation

Every programming language reflects an underlying *model for abstraction*. For example, an imperative language relies upon the procedure, which is simply a composition of commands that act upon objects to update their state. The procedure is itself a command, and may be subject to further composition. Operators dictating the manner of composition for both command and object, together with a set of primitives for each, complete the model.

Three questions arise over any proposed language and its associated model for abstraction: Can the model directly capture all the behaviour of the systems we may wish to build? Is each program then *transparent*? How precisely is the meaning of each program defined, and can we relate one program to another? To put it more simply, how easy are programs both to write and to read, and how can we tell, for example, that one may perform a function indistinguishable from another?

Early languages complicated their model by including reference to the machine level, with pointer and 'goto'. Despite the fact that it has been repeatedly shown to be unnecessary, such things remain in contemporary languages, denying simplicity and transparency of expression. These languages also typically lack a formal semantics, despite valiant attempts to install a foundation after the house has been built. Finally, they fail to capture concurrent and reactive behaviour, or do so in a manner that is unnecessarily complicated and obscure.

For many, occam [1] offered a remedy for all these shortcomings. Its *process-oriented* model could boast a formal foundation in the theory of Communicating Sequential Processes (CSP) [2,3], affording analysis, and a precise interpretation, of every program.

While occam arguably represented a major advance, it left room for further progress. In some ways, it was perhaps too simple. An object could only be copied, and not passed, between processes, giving rise to some inefficiency. Only processes, and not objects, could en-

capsulate behaviour with information, limiting choice in system abstraction, and suggesting a conflict with (highly popular) object-oriented design. There was no facility for collecting definitions together, according to their utility – something we shall refer to here as *project*, as opposed to system, modularity – in contrast with Java, for example.

Finally, occam offered no intrinsic help in avoiding the additional pathological behaviour inherent with concurrent and reactive behaviour. In particular, one had to rely upon informal design patterns to exclude the threat of deadlock.

The Honeysuckle Programming Language

The Honeysuckle project began as an attempt to establish a successor to occam that builds on its strengths [4]. In particular, it overcomes the additional threat of pathological behaviour introduced when incorporating concurrency and prioritized reactive (event-driven) behaviour. Honeysuckle incorporates formal design rules that guarantee every design and program *a priori* deadlock-free [5]. These are verified 'statically' (upon compilation).

In the Honeysuckle model for abstraction, a system may be reduced into components, which, as in occam, are processes. Unlike occam, the interface between components is expressed as *services* provided or consumed. A service comprises a sequence of communications, each defined by the type of object conveyed (if any), whether ownership, or simply a copy, is passed, and the direction in which conveyance takes place. Each service is oriented according to initiation, which is performed by the client (consumer). At the "sharp end" is the server (provider). A component may either uphold mutual exclusion or *alternate* provision of multiple (aggregated) services according to static prioritization.

It is worth noting that a service is ultimately implemented using at most two synchronous *channels* (one in each direction), and so any Honeysuckle program could be expressed occam. Service architecture may therefore be considered 'superstructure'.

Like occam, Honeysuckle has been built upon a formal foundation, derived from CSP. A formal definition has been provided of both service and *service network component* (SNC), on which a proof of deadlock-freedom has been based [5].

While a component may be specified by its interface, its design is expressed via the declaration of a service network. Interface definition and network declaration form the *context* in which an implementation is later described, and are rendered explicit using a subset of Honeysuckle – the *Honesuckle Design Language* (HDL) – as a 'header' above a process implementation. For example:

```
definition of process p

  ... imports

  process p is
  {
    interface
      provider of s1
      client of s2

    network
      s1 > s2
  }
```

Here, a component is defined that provides one service while consuming another. The service provided *depends* upon the one consumed. Dependency is one way in which services may inter-relate. Each 'input' (service provided at the component interface) may relate to 'output' (service consumed at the interface) via a *chain of dependency*.

While it is possible to express the behaviour of any system solely in terms of service architecture, implementation of every dependency requires the definition of a process.

Dependency *abstracts* process for the purpose of design.

Note that no implementation is given in the above definition. Design and implementation are intended to be separate, with each design verified against rules which guarantee deadlock-freedom [5]. An implementation may be provided later, and is verified only against a design.

Three kinds of *item* must be defined in a Honeysuckle program – process, service, and object class. Definitions may be collected, according to their utility, and traded (imported and exported). The ability to form a *collection* affords "project modularity" – the possibility of sharing (reusing common) definitions between distinct projects.

A Honeysuckle implementation is imperative and process-oriented, and must describe *how* each service is provided or consumed, using the same primitives as occam – SEND, RECEIVE, and ASSIGN – plus two more – TRANSFER and ACQUIRE – which refer to the conveyance of an object, rather than a value. (The distinction between value and object (reference) is made by choice of operator, rather than by modifying an identifier.)

Each process is constructed in much the same way as Pascal, using sequence, selection, and repetition, but with the addition of a parallel construct, as in occam, and one that affords *prioritized alternation*. The latter dictates pre-emptive behaviour similar to that brought about by hardware prioritized vectored interrupts, but subject to a certain discipline [6].

Honeysuckle syntax is intended to result in transparent (highly readable) programs, where simple function is achieved with simple text. For example, a "Hello world!" process can be expressed in just two lines, free of any artifact:

```
process hello is
  send "Hello world!" to console
```

Here, 'console' is the name of a service, not a process.

Overall, it is hoped that Honeysuckle offers even greater simplicity, efficiency, and transparency than did occam, while introducing *a priori* security against deadlock.

The Need for Refinement of Honeysuckle and the PSA *Model for Abstraction*

Honeysuckle incorporates a model for system abstraction that allows a design to be expressed entirely in terms of communication, without reference to process or object. As stated above, the intention is that such a design should offer sufficient information for deadlock to be excluded *before* any implementation is conceived. Unfortunately, with the language thus far reported [7], it remains possible to express a design which would endorse some incorrect implementations – ones that would fail verification of the conditions governing the behaviour of a component of a deadlock-free network [5]. The simplest example is the classic one of "crossed servers", where two clients each seek sequential access to two servers. This would form a hitherto legitimate Honeysuckle design, though any implementation would break at least one condition governing the behaviour of a service network component.

As a result, it became clearly desirable to establish an enhanced service-oriented model in which one could distinguish the correct from the incorrect. Changes to the Honeysuckle Design Language (HDL) are proposed that allow every legitimate design to dictate an implementation that will satisfy all conditions for deadlock-freedom. These changes are documented in Section 1 and 2. Sadly, this does entail complicating the language, but the prize of properly separating verifiable design from implementation is believed well worthwhile.

Interleaving in Context

The refinements to the PSA model needed to address the deficiency mentioned above depend upon a notion of 'interleaving'. In CSP, 'interleaving' refers to that of events identified with

processes. Here, we refer to events (communications) identified instead with *services*.

For example, suppose a service comprises the sequence $\{c_i^1\}$, where i indexes progression in the delivery of that service. A second service might be formed by the sequence $\{c_j^2\}$, with progress indexed by j. Should the two services progress together, we might then observe a sequence $c_1^1, c_1^2, c_2^1, c_2^2, \ldots$.

Whenever one service starts before the other is complete, we say that the two *interleave*.

There are three different ways in which services may combine and interleave. Each is capable of endorsing conflict with the formal conditions laid down upon service and component [8,5], and thus aberrant behaviour. To remove such conflict, and establish the possibility of a guarantee of deadlock-freedom, given only a PSA design, certain attributes are introduced to each form of service combination, according to whether interleaving is allowed or denied.

These circumstances and changes are described in Section 2 below.

A Visual Design Language

While investigating these changes, it became increasingly apparent that design could be expressed with considerably greater transparency using pictures. Greater transparency is a key aim in the development of Honeysuckle and the PSA model for system abstraction.

Visual programming languages were explored widely in the 1970s. Particularly successful was the notion of "visual data flow", where processes embody functions repeatedly executed. Each one would await reception on each input port, evaluate the appropriate function, and then transmit the result. (Multiple output ports implied multiple functions evaluated.)

Evidence of the utility of this idea can be found in the huge success of LabView^{TM}[1], which facilitates the visual programming of industrial test instrumentation. The dominant commercial appeal appears to be productivity. This is widely believed to result from the visual nature of the language and the consequent ease of expression. Productivity, through ease of expression, is another key aim in the development of PSA and Honeysuckle.

Because it would increase the ease with which a design may be both written and read, a Honeysuckle Visual Design Language is proposed and is described in Section 3.

Summary

Changes to the (textual) Honeysuckle Programming Language (HPL) are described first. Section 1 addresses changes to the abstraction, and syntax for the description, of the component interface. Section 2 deals with changes in the way an internal prioritized service network is conceived and expressed. Finally, Section 3 describes the proposed Honeysuckle Visual Design Language (HVDL).

1. Concurrency and synchronization in the component interface

1.1. Distributed Service Provision

To express the interface of a component, only services provided and consumed need be declared, and not their interdependency. It may be possible for any service offered to be consumed by more than one component simultaneously. Provision must then be *distributed* across more than one internal process [7]. An interface declaration should include the degree of concurrency possible. For example:

```
interface
  provider of [4] publisher
```

[1] LabView is a trademark of National Instruments Inc.

In the interest of brevity, graphical expression of the component interface is not elaborated here, though it is similar to that employed to depict design, described later.

1.2. Synchronization of Shared Provision

Any service provided by a component may always be *shared* between multiple consumers. There is no need to advertise the fact, or limit the degree. As the number of consumers rises, beyond the degree of concurrency available, the delay will simply increase.

A queue will form, and service will become sequential.

When sequential service is *synchronized* [9], it is to be shared between a specific number of clients. Each will be served once before the cycle can begin again.

Note that no provision can be both concurrent *and* synchronized, in Honeysuckle.

The description of a component interface is a little different to that of each connection within a network. Because of the possibility of dependency, synchronization may encompass multiple distinct services. Fig. 13 shows an example of how this can occur. The corresponding interface can be expressed as follows:

```
interface
  synchronized
    [2] provider of claim
    provider of request
```

At an interface, it could conceivably make sense for a provision to be both synchronized *and* distributed. One purpose behind synchronization is to expose multiple processes to state whose evolution is perceived in common. Distributed providers could perhaps relay such common state via a single shared service on which they all depend. However, such access would again be sequential, even if such dependency were somehow enforced. Hence, mutual exclusion between distributed (concurrent) and synchronized (sequential) service is to be maintained, in Honeysuckle, at the interface, as within the network.

Once an interface has been established, a design can be developed, which requires the definition of a prioritized service network, declared separately.

2. Interleaving of Internal Service Provision, Dependency, and Consumption

2.1. Interleaved Provision

The Honeysuckle Design Language (HDL) provides a construct by which to express the prioritized interleaving of service *provision* [7]. Delivery of one service may be pre-empted by another, should a client initiate it.

```
network
  interleave
    s1
    s2
```

In the above, should provision of $s2$ be underway when $s1$ is initiated, it will cease to progress until $s1$ provision is complete, whereupon it will resume. Delivery of $s2$ is thus interrupted and pre-empted by that of $s1$. There is no intrinsic necessity for this condition. It has been adopted in order to simplify the model for abstraction, and to remove problems which otherwise arise in forming networks, secure against pathological behaviour.

Note that this behaviour can be stated wholly in terms of *service* provision.

Interleaved provision may be directly implemented using prioritized alternation (the WHEN construct) [6]. Only in implementation do we express behaviour in *process*-oriented terms. Note that alternation is *reactive* (event-driven), but free of concurrency. No two processes composed in this way run concurrently at any time.

Given the semantics applied to interleaved service provision, and its implementation via prioritized alternation, all the conditions placed upon both service conduct and each service network component can be maintained, except for systems where there is feedback.

With feedback, a service of lower priority must *depend* upon one of higher priority.

2.2. Interleaved Dependency

When the provision of one service depends upon the consumption of another, we say that a *dependency* exists. This relation may recur, allowing the formation of a *chain of dependency* of arbitrary length. For example:

```
network
  s1 > s2
  s2 > s3
  ...
```

A chain of dependency may form a feedback path across interleaved provision. To indicate this textually, a hyphen is introduced to the right of the service with the fed-back dependency, and to the left of the one on which it depends. For example:

```
network
  interleave
   -s1
    s2 > s1-
```

Here, provision of $s2$ depends upon consumption of another service ($s1$) with which provision is interleaved, and which has higher priority .

There exists the potential for conflict between the condition upon interleaved provision – that the interrupting service be completed before the interrupted one resumes – and any interleaving across the dependency. Should the consumption of $s2$ interleave with the provision of $s1$ then the former would have to continue while the latter proceeds. As a consequence, we need to refine our notion of prioritized service architecture (PSA) for consistency to prevail.

We shall therefore distinguish two kinds of dependency.

An *interleaved* dependency is one where communications of the service consumed interleave with those of the one provided. An *interstitial* dependency occurs where a service is consumed in its entirety between two communications of the one provided.

Since interstitial dependencies are the safer of the two, they will be assumed, by default. An interleaved dependency must now be indicated, with additional notation. For example:

```
s1 > s2, |s3
```

Here, provision of $s1$ has two dependencies; one interstitial, and one interleaved ($s3$).

The previous interleaving with feedback is now both more precisely defined and free of any conflict between applicable conditions. For the same to be said of any interleaved provision, a simple additional design rule must apply:

Every feedback chain must contain at least one interstitial dependency.

A feedback chain of length one has a natural service-oriented interpretation, though is a little troublesome in implementation as it imposes a measure of concurrency upon provider and consumer.

2.3. Interleaved Consumption

Having discussed interleaved provision and interleaved dependency, we now turn to the third and final occasion for interleaving – interleaved *consumption*.

Let us suppose we have a dependency upon two services, as described previously, but where these services are in turn provided subject to mutual exclusion [7][2]:

```
network
  s1 > s2, |s3
  select
    s2
    s3
```

If consumption of $s2$ and $s3$ interleaves then deadlock will surely ensue — a classic example of "crossed servers". For example, an attempt to initiate $s3$ might be made before $s2$ is complete. That attempt will fail because provision of $s2$ and $s3$ has been declared mutually exclusive. On the other hand, if we already know that $s2$ is completed before $s3$ is begun, the configuration is entirely safe.

To ensure security, and retain pure service-oriented program abstraction, we therefore need to distinguish between interleaved and *sequential* consumption. An alternative to the above that is secure against deadlock would be expressed as follows:

```
network
  s1 > s2; |s3
  select
    s2
    s3
```

A semicolon has replaced the comma, indicating that consumption of $s3$ commences only after that of $s2$ is complete.

A comma does not necessarily indicate the converse – that consumption of the two services interleaves. Instead, it merely indicates the *possibility* of interleaving, which would be enough, in this case, to deny any guarantee of the absence of deadlock.

One other potential conflict with security against deadlock arises when chains of dependency cross. A design rule [5] requires that they never do so when fed forward to subsequent interleaved provision (inverting priority). A refinement of this rule is needed to account for crossing over by a feedback chain. We require that any dependency additional to any link in the feedback chain be consumed in sequence, and not interleaved.

3. The Honeysuckle Visual Design Language (HVDL)

3.1. Philosophy

Design, in general, refers to the decomposition of a system into components. Its purpose is to facilitate both analysis and separated development and test. Its outcome must therefore include a precise and complete description of component function and interface. With Honeysuckle, this need refer only to the pattern of communication between components.

PSA is particularly well-suited to *graphical* construction and editing, which is believed capable of improving both insight and productivity. A suitable development tool is needed to both capture a design and, on command, translate it into textual form. In no way does such a tool replace a compiler. It will merely act as an aid to design, though will be capable of verifying, and thus enforcing, both principal design rules [5].

Systems and components, defined purely according to services provided and consumed, may be composed or decomposed, without reference to any process-oriented implementation. Each *dependency* may be reduced, or combined with another. A HVDL editor is expected to afford either operation and thus allow the description of any service architecture by step-

[2]The keyword SELECT has been substituted for EXCLUSIVE, used in the earlier publication.

wise refinement. Only when an engineer decides that a system is *fully defined* (*i.e.* ready for implementation in HPL) is a translation into textual form carried out.

HVDL thus constitutes an interactive tool for the designer, whereas HPL serves the implementer (programmer). Design is bound by the Honeysuckle design rules (verified continually by HVDL), while any implementation remains bound by the design.

3.2. Independent Service

A simple, self-contained, system might comprise just the provision and consumption of a single service (Fig. 1).

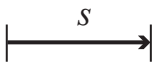

Figure 1. A single service, both provided and consumed internally.

An arrow represents availability of the service s, and is directed towards the provider.

Provision of s is termed *independent*, by which we mean that it depends upon consumption of no other service. The nature of service s is documented separately.

The thin line at either end of the arrow denotes internal consumption. As depicted, the system therefore has no interface. External consumption or provision may be indicated by the same thin line drawn *across* an arrow, close to one end or the other accordingly (Fig. 2).

Figure 2. External consumption (left) and external provision (right).

Within the corresponding textual description, the distinction is rendered explicit within an accompanying component INTERFACE section [9].

3.3. Dependency and Interface

When provision of one service is dependent upon the consumption of another, an *interface* exists between services, denoted graphically by a '•'. While a process will be required to implement such an interface, there is no need to consider its nature when documenting design.

The nature of the dependency falls into one of two distinct categories (Fig. 3).

Figure 3. A single interstitial (left) and interleaved (right) dependency.

The distinction between interstitial and interleaved dependency was discussed earlier, in Section 2.2. Note that this forms an attribute of the interface, and not of either service.

Dependency upon multiple services requires a distinction between interleaved and sequential consumption (2.3). The latter is indicated graphically by the presence of a thin dividing line (Fig. 4). Absence of such a line indicates the *possibility* of interleaving.

In Fig. 4 (right), the vertical bar, just to the right of the interface symbol, confirms interleaving. Since both consumed services are interstitial, this implies that the two are completed within the same interstice (interval between two communications of the dependent service).

Figure 4. Sequential (left) and interleaved (right) consumption.

3.4. Aggregated Service Provision

As we described in Sections 2.1 and 2.3, service provision may be combined in two ways — mutually exclusive *selection*, and prioritized (pre-emptive) *interleaving* (Fig. 5).

Figure 5. Exclusive selection with shared dependency (left), and prioritized interleaved provision (right).

With exclusive (selective) provision, any dependency of any member is also a dependency of the entire *bunch*. This is because any member may be forced to await consumption. However, it remains useful still to indicate which among the bunch of services possesses a direct dependency. Alignment with the interface connection, or a short horizontal line extending an arrow, is enough (Fig. 6).

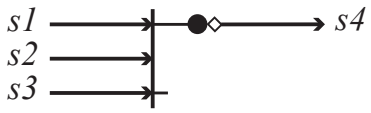

Figure 6. Indicating which mutually exclusive services are directly dependent.

Mutually exclusive selection is both commutative and associative. To eliminate redundant options for expression, nesting is denied. Offering the same service twice is similarly redundant, given *shared provision* (see below), and is thus also denied.

Multiple dependencies, within an interleaved provision, are drawn by extending the corresponding interface vertically (Fig. 7).

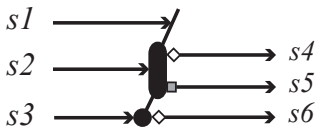

Figure 7. Multiple dependencies within interleaved provision.

Nesting of one INTERLEAVE construction inside another is unnecessary and denied. SELECT within INTERLEAVE is permitted, but not INTERLEAVE within SELECT.

In Section 2.3, we met the constraint on feedback (Fig. 8) across an interleaving that demands that the feedback chain ends with higher-priority provision. A second condition also applies; that, in any feedback chain, at least one dependency must be interstitial. When

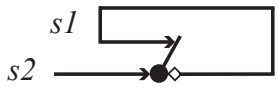

Figure 8. Feedback across interleaved provision.

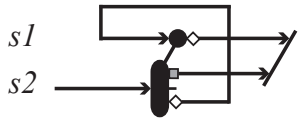

Figure 9. An example of legitimate consumption sequential to a feedback path.

a feedback chain has length one, the dependency is necessarily interstitial, and a measure of concurrency is required between consumer and producer, in implementation.

Any additional dependency of the lower-priority service must be arranged in sequence with the feedback path, when fed forward to further interleaved provision, as in Fig. 9. The order of the sequence is immaterial.

3.5. Sharing and Distribution

Shared provision is depicted via the convergence of multiple arrow shafts towards a single arrow head (Fig. 10).

Figure 10. Shared (left), and synchronized shared (right), provision.

Honeysuckle also affords *synchronized* shared provision, which requires each client to consume the service concerned once before the cycle can begin again [9]. Synchronized sharing is indicated graphically by a loop where arrow shafts join.

Like sharing, synchronized sharing is not an attribute of either service or process. It refers only to how a service is consumed within a particular network design.

While sharing offers many-to-one interconnection, *distributed* shared provision affords many-to-many configurations (Fig. 11).

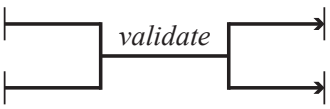

Figure 11. Distributed shared provision.

It can never make sense to offer more providers than consumers, and so the very possibility is denied. There are therefore no one-to-many connections within any service digraph.

Multiple providers of a dependency may also be expressed (Fig. 12).

All distributed providers must be fully independent of their consumer. No feed-back is allowed. Behaviour is unaffected when a dependency is shared and/or distributed.

Synchronized sharing can only delay, and not deny, consumption.

Both graphical and textual expression of sharing and distribution renders the degree of each explicit. (Recall that, barring asymmetry, sharing denies the need for provision of

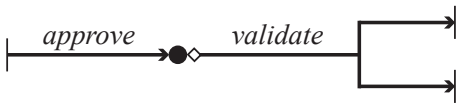

Figure 12. Distributed provision of a dependency.

multiple instances of the same service [7].) The number of clients to a shared service need only be revealed in the description of a complete system. *Synchronized* sharing is ultimately an exception to this rule, since behaviour might depend upon, or limit, its degree.

An example might be found in a component that protects a shared resource by requiring a claim by, say, three separate clients prior to release (Fig. 13).

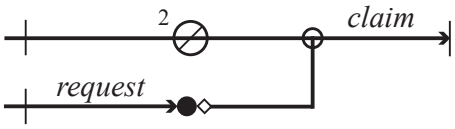

Figure 13. Synchronized sharing for resource protection.

Textual expression of such a system, though concise, is far less transparent:

```
network
  request > claim
  synchronized [3] shared claim
```

Note that an interface declaration says nothing about dependency of 'input' (service provision) upon 'output' (service consumption). As a result, it is impossible to capture any higher level of abstraction than the one implemented. Graphical expression, using HVDL has no such limitation.

An explicit indication of multiplicity in service provision can have two distinct meanings. As we have seen in Fig. 13, it can refer to the number of instances that must be synchronized. (It remains possible for a single client to satisfy this constraint, though perhaps the most common application will be that each instance will be consumed by a separate process.)

In addition, multiplicity can refer to the degree of distributed provision, which equates with the degree of concurrency possible in consumption. An example is depicted in Fig. 14. Here, a system provides up to three concurrent instances of a service. A fourth request will incur a delay until a provider becomes available.

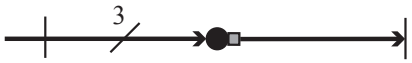

Figure 14. Distributed (concurrent) provision, with a single internal shared dependency.

Note that provision of the dependency is not similarly replicated, and is therefore shared.

4. System Abstraction using HVDL

4.1. Introduction

Honeysuckle was previously able to abstract a system by decomposition into a network of processes, communicating under service protocol. HVDL does nothing to change that. Its

effect is rather to provide the means of progressively refining system abstraction purely in terms of priioritized service architecture (PSA). As such, it represents a powerful design tool.

A designer might begin by simply specifying a system via its input and output connections, together with their interdependency. They might subsequently refine this (perhaps by clicking on the graphical representation of an interface) to achieve reduction. At any time, they may choose to subdivide the system into separated components. Refinement might continue until implementation becomes possible. At this point, a command can be given to generate both interface and network definition for each and every component. (A HVDL editor is expected to maintain a description of system/component composition.)

Advantages of this approach include the ability to maintain a hierarchical and graphical depiction of a concurrent/reactive system with any degree of complexity, in a manner that can be guaranteed authentic. Because a HPL compiler must guarantee conformance with interface and network definition, any changes to the abstract system/component specification or design can be similarly guaranteed in the implementation.

Of course, freedom from any possibility of deadlock also remains guaranteed.

4.2. Component Specification and Refinement

A component interface is expressed, using either HDL or HVDL, without including any dependency between 'input' (service provision) and 'output' (consumption). Only information required for connection to other components is provided. Any provision may be shared, but the degree to which it is either synchronized or distributed must be noted. Note that, because of internal structure (*e.g.* Fig. 13), synchronization may extend over services that are distinct, as well as shared. Distributed provision affords concurrent consumption.

Design might begin with only the highest level of abstraction – that of the dependency between component provision and consumption. Refinement might consist of either directly replacing an interface with additional structure (an *expansion*) or of naming that interface and giving it a separate description (an *embedding*).

It is not only an interface that can see its definition refined. A *terminal* (internal provision or consumption) may also be subject to either expansion or naming and embedding.

4.3. An Example

For illustration, let us consider a modern "print on demand" (POD) publishing system, that serves both customers and authors, and which consumes a printing service.

Our design will look a lot more like that of hardware than of software, and will omit certain parameters important in real life, such as the time taken either to complete a service or to progress between steps within it. However, as with a hardware (digital system) design, every detail required for the addition of such constraints to the specification is included.

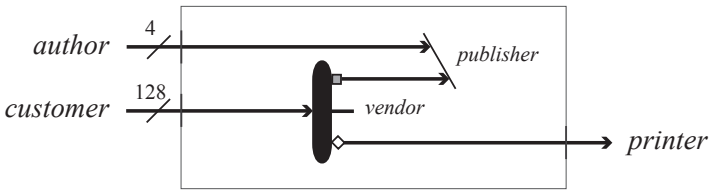

Figure 15. An abstraction of a "print on demand" publishing system.

Fig. 15 depicts an initial design with 'input' and 'output' ports labelled. Such a system can be defined via the following interface:

```
interface
  provider of [128] customer
  provider of [4] author
  client of printer
```

A convention is helpful whereby services are named according to provider. This affords readable use of the Honeysuckle primitives SEND, RECEIVE, TRANSFER, and ACQUIRE [10]. For example:

```
send draft to publisher
```

Client connections ('input' ports) are named only for illustration here. All that is significant is the name of the service provided, and the degree of concurrency available. Up to 128 customers can be served concurrently before one has to wait their turn. A much lower degree of concurrent provision is appropriate for authors.

Fig. 15 depicts the design (prioritized service architecture) as well as the specification (component interface). Labels appropriate to a process implementation have been added. The vendor interface exhibits a dependency of the customer service upon sequential consumption of publishing and printing services. A publisher process alternates service provision between authors and vendor. Note that serving the vendor is attributed a higher priority. (In this illustration, priority increases downwards. As a result, *every* graphic depicting an interleaving must lean to the left. Consistency, in this respect, is essential.)

Before declaring a network, we shall first refine our initial design. Clearly, management of provision (vending), consumption (printing and despatch), and internal accounting and record-keeping, will require significant decomposition into internal services (Fig. 16).

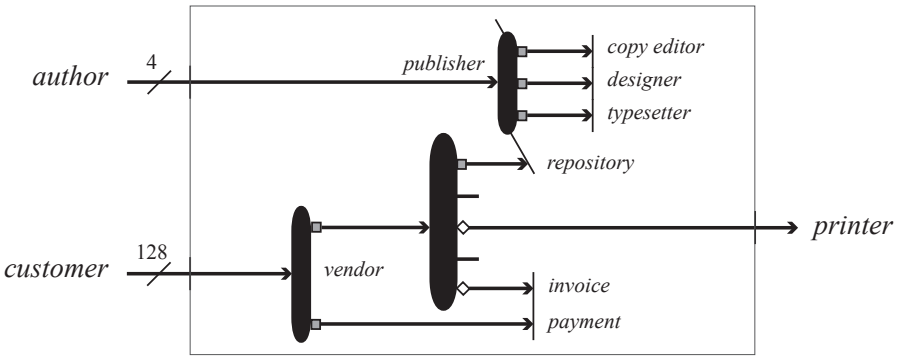

Figure 16. A refinement to the POD system where distribution has been separated.

It may eventually be considered worthwhile to "out-source" production (typesetting, copy-editing, cover and block design, *etc.*). Our design can thus be expressed via the following NETWORK statement:

```
network
  customer > |supply, |payment
  supply > |repository; printer; invoice
  interleave
    recover
    author > |copy_editor, |designer, |typesetter
```

Only prioritized service architecture (PSA) has been required in the description of a design. There has been no need to identify any process, though their location is self-evident.

5. Conclusion

Another step has been taken in the design of a pure service-oriented model for system abstraction, reflecting *protocol* rather than process, together with a corresponding programming language. The predominant aims of easing decomposition and denying pathological behaviour are well-served by an emphasis on communication, which both affords the transparent description of the individual component interface and exposes the overall pattern of interpendency within a complete network, allowing a measure of holistic analysis.

The Honeysuckle Programming Language (HPL) is intended to guarantee conformance with the conditions previously laid down for systems with prioritized service architecture (PSA), which in turn guarantee freedom from deadlock [5]. To this end, it has been necessary to add a little complication to the Honeysuckle Design Language (HDL – a subset of HPL). While HDL is a little less simple than before, it remains transparent and intuitive.

A later paper will offer a formal proof that conformance with the conditions follows conformance with the language, once the latter is finally complete and stable. No significant difficulty is anticipated here since those conditions that apply to each service network component (process) are easily adapted to instead refer to each service interface. No reference to any (process-oriented) implementation is necessary.

A *graphical* counterpart to HDL – the Honeysuckle Visual Design Language (HVDL) has been presented. This offers an alternative to textual expression that affords much greater transparency. A suitable graphical editor could offer hierarchical decomposition (or composition) of systems with PSA, and the automatic generation of interface and network definition that can be subsequently implemented using HPL. It could also offer visualization of existing textual designs, since a one-to-one correspondence exists between text and graphics.

HVDL demonstrates the possibility of employing a formal model for component-based systems to create development tools that combine high productivity with high integrity, free of the need for staff with a scarce combination of mathematical and engineering skill. It also demonstrates the potential for system definition and design using a purely communication-oriented vocabulary and syntax.

Applicability to the engineering of distributed systems with service architecture is obvious, but HDL is by no means limited to such "high-level" abstraction. PSA, and thus Honeysuckle, is equally applicable to engineering systems at *any* level of abstraction, from hardware upward, and seems particularly well-suited to high-integrity embedded systems, and perhaps to 'co-design' of systems to be implemented with a mixture of hardware and software.

References

[1] Inmos. *occam 2 Reference Manual*. Series in Computer Science. Prentice Hall International, 1988.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall International, 1985.

[3] A. W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice-Hall, 1998.

[4] Ian R. East. Towards a successor to occam. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, *Proceedings of Communicating Process Architecture 2001*, pages 231–241, University of Bristol, UK, 2001. IOS Press.

[5] Ian R. East. Prioritised Service Architecture. In I. R. East and J. M. R. Martin et al., editors, *Communicating Process Architectures 2004*, Series in Concurrent Systems Engineering, pages 55–69. IOS Press, 2004.

[6] Ian R. East. Programming prioritized alternation. In H. R. Arabnia, editor, *Parallel and Distributed Processing: Techniques and Applications 2002*, pages 531–537, Las Vegas, Nevada, USA, 2002. CSREA Press.

[7] Ian R. East. Concurrent/reactive system design with honeysuckle. In A. A. McEwan, S. Schneider, W. Ifill, and P. H. Welch, editors, *Proceedings of Communicating Process Architecture 2007*, pages 109–118, University of Surrey, UK, 2007. IOS Press.

[8] Jeremy M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, Hunter Street, Buckingham, MK18 1EG, UK, 1996.

[9] Ian R. East. Interfacing with Honeysuckle by formal contract. In J. F. Broenink, H. W. Roebbers, J. P. E. Sunter, P. H. Welch, and D. C. Wood, editors, *Proceedings of Communicating Process Architecture 2005*, pages 1–12, University of Eindhoven, The Netherlands, 2005. IOS Press.

[10] Ian R. East. The Honeysuckle programming language: An overview. *IEE Software*, 150(2):95–107, 2003.