Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

Interfacing with Honeysuckle by Formal Contract

Ian EAST

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

ireast@brookes.ac.uk

Abstract. Honeysuckle [1] is a new programming language that allows systems to be constructed from processes which communicate under service (client-server or master-servant) protocol [2]. The model for abstraction includes a formal definition of both service and *service-network* (system or component) [3]. Any interface between two components thus forms a binding contract which will be statically verified by the compiler. An account is given of how such an interface is constructed and expressed in Honeysuckle, including how it may encapsulate state, and how access may be shared and distributed. Implementation is also briefly discussed.

Keywords. Client-server protocol, compositionality, interfacing, component-based software development, deadlock-freedom, programming language.

Introduction

The Honeysuckle project has two motivations. First, is the need for a method by which to design and construct reactive (event-driven) and concurrent systems free of pathological behaviour, such as deadlock. Second, is the desire to design a new programming language that builds on the success of occam [4] and profits from all that has been learned in two decades of its use [5].

occam already has one worthy successor in occam-π which extends the original language to support the development of distributed applications [6]. Both processes and channels thus become mobile. Honeysuckle is more conservative and allows only objects mobility. Emphasis has instead been placed on securing integrity within the embedded application domain. Multiple offspring are testimony to the innovative vigour of occam.

Any successor must preserve its salient features. occam facilitates the natural expression of concurrency without semaphore or monitor. It possesses transparent, and mostly formal, semantics, based upon the theory of Communicating Sequential Processes (CSP) [7,8]. It is also *compositional*, in that it is rendered inherently free of side-effects by the strict separation of value and action (the changing of value).

occam also had its weaknesses, that limited its commercial potential. It offered poor support for the expression of data structure and none for dynamic (abstract) data types. While processes afford encapsulation and allow effective system modularity, there is also no support for project (source code) modularity. One cannot collect related definitions in any kind of reusable package. Also, the ability only to copy a value, and not pass access to an object, to a parallel process caused inefficiency, and lay in contrast with the passing of parameters to a sequential procedure.

Perhaps the most significant factor limiting the take-up of occam has been the additional threats to security against error that come with concurrency; most notably, deadlock. Jeremy Martin successfully brought together theoretical work on deadlock-avoidance using CSP with the effective design patterns for process-oriented systems introduced by Peter Welch et al.

[9,10,11,12]. The result was a set of *formal design rules*, each proven to guarantee deadlock-freedom within a CSP framework.

By far the most widely applicable design rule relies on a formal *service* (client-server) protocol to define a model for system architecture. This idea originated with Per Brinch-Hansen [2] in the study of operating systems. Service architecture has a wide domain of application because it can abstract a large variety of systems, including any that can be expressed using *channels*, as employed by occam. However, architecture is limited to hierarchical structure because of a design rule that requires the absence of any directed circuit in service provision, in order to guarantee freedom from deadlock.

A formal model for the abstraction of systems with service architecture has been previously given [3], based upon the rules employed by Martin. This separates the abstraction of service protocol and service network component, and shows how the definition of system and component can be unified (a point to be revisited in the next section). Furthermore, the model incorporates *prioritisation*, which not only offers support for reactive systems (that typically prioritise event response), but also liberates system architecture from the constraint of hierarchical (tree) structure. Finally, a further proof of the absence of deadlock was given, subject to a new design rule.

Prioritised service architecture (PSA) presents the opportunity to build a wide range of reactive/concurrent systems, guaranteed free of deadlock. However, it is too much to expect any designer to take responsibility for the *static verification* of many formal design rules. Specialist skills would be required. Even then, mistakes would be made. In order to ease design and implementation, a new programming language is required. The compiler can then automate all verification.

Honeysuckle seeks to combine the ambition for such a language with that for a successor to occam. It renders systems with PSA simple to derive and express, while retaining a formal guarantee of deadlock-freedom, without resort to any specialist skill or tool beyond the compiler. Its design is now complete and stable. A compiler is under construction and will be made available free of charge.

This paper presents a detailed account of the programming of service protocol and the construction of an interface for system or component in Honeysuckle. In so doing it continues from the previous language overview [1]. We begin by considering the problem of modular software composition and the limitations of existing object- and process-oriented languages.

1. The Problem of Composition

While occam is compositional in the construction of a monolithic program, it is *not* so with regard to system modularity. In order to recursively compose or decompose a system, we require:

- some components that are indivisible
- that compositions of components are themselves valid components
- that behaviour of any component is manifest in its interface, without reference to any internal structure

Components whose definition complies with all the above conditions may be termed *compositional* with regard to some operator or set of operators. As alluded to earlier, it has been shown how service network components (SNCs) may be defined in such a way as to satisfy the first two requirements when subject to parallel composition [3].

A corollary is that any system forms a valid component, since it is (by definition) a composition. Another corollary, vital to all forms of engineering, is that it is then possible to *substitute any component with another*, possessing the same interface, without affecting either

design or compliance with specification. Software engineering now aspires to this principle [13].

Clearly, listing a series of procedures, with given parameters, or a series of channels, with associated data types, does little to describe object or process. To substitute one process with another that simply sports the same channels would obviously be asking for trouble. A much richer language is called for, in which to describe an interface.

One possibility is to resort to Floyd-Hoare logic [14,15,16] and impose formal pre- and post-conditions on each procedure ('method') or channel, and maintain invariants associated with each component (process or object class). However, this would require effectively the development of a language to suit each individual application and is somewhat cumbersome and expensive. It also requires special skill. Perhaps for that reason, such an explicitly formal approach has not found favour in much of industry. Furthermore, no other branch of engineering resorts to such powerful methods.

Meyer introduced the expression *design by contract* [17], to which he devotes an entire chapter of his textbook on object-oriented programming [18]. This would seem to be just a particular usage of invariants and pre- and post-conditions, but it does render clear the principle that some protocol must precede composition and be verifiable.

The difficulty that is peculiar to software, and that does not apply (often) to, say, mechanical engineering, is, of course, that a component is likely to be capable of complex behaviour, responding in a unique and perhaps extended manner to each possible input combination. Not many mechanical systems possess memory and the ability to change their response in perhaps a highly non-linear fashion. However, many electronic systems do possess significantly complex behaviour, yet have interfaces specified without resort to full first-order predicate calculus. Electronic engineers expect to be able to substitute components according to somewhat more specific interface description.

One possibility for software component interface description, that is common with hardware, is a formal communication protocol detailing the order in which messages are exchanged, together with their type and structure. In this way, a binding and meaningful contract is espoused. Verification can be performed via the execution of an appropriate "state-machine" (finite-state automaton (FSA)).

Marcel Boosten proposed just such a mechanism to resolve problems encountered upon integration under component-based software development [19]. These included race conditions, re-entrant call-backs, and inconsistency between component states. He interposed an object between components that would simulate an appropriate FSA.

Communication protocol can provide an interface that is both verifiable and sufficiently rich to at least reduce the amount of logic necessary for an adequate definition, if not eliminate it altogether.

In Honeysuckle, an interface comprises a list of *ports*, each of which corresponds to one end (*client* or *provider*) of a service and forms an attribute of the component. Each service defines a communication protocol that is translated by the compiler into an appropriate FSA. Conformance to that protocol is *statically* verifiable by the compiler.

Static verification is to be preferred wherever possible for the obvious reason that errors can be safely corrected. Dynamic verification can be compared to checking your boat *after* setting out to sea. Should you discover a hole, there is little you can then do but sink. Discovering an error in software that is deployed and running rarely leaves an opportunity for effective counter-measures, still less rectification. Furthermore, dynamic verification imposes a performance overhead that may well prove significant, especially for low-latency reactive applications.

It is thus claimed here that (prioritised) service architecture is an ideal candidate for secure component-based software development (CBSD).

Honeysuckle also provides *balanced abstraction* between object and process. Both static and dynamic object composition may be transparently expressed, without recourse to any explicit reference (pointer). Distributed applications are supported with objects mobile between processes. Together, object and service abstraction affords a rich language in which to express the interface between processes composed in either sequence or parallel.

2. Parallel Composition and Interfacing in Honeysuckle

2.1. Composition and Definition

Honeysuckle interposes "clear blue water" between system and project modularity. Each definition of process, object, and service, is termed an *item*. Items may be gathered into a *collection*. Items and collections serve the needs of separated development and reuse.

Processes and objects are the components from which systems are composed, and together serve the needs of system abstraction, design, and maintenance. Every object is owned by a single process, though ownership may be transferred between processes at run-time. Here, we are concerned only with the programming of processes and their service interface.

A program consists of one or more item definitions, including at least one of a process. For example:

```
definition of process greet

  imports
    service console from Environment

  process greet :
  {
    interface
      client of console
    defines
      String value greeting : "Hello world!\n"

    send greeting to console
  }
```

This defines a unique process *greet* that has a single port consuming a service named *console* as interface. The console service is assumed provided by the *system environment*, which is effectively another process composed in parallel (which must include "provider of console" within its interface description). Figure 1 shows how both project and system modularity may be visualized or drawn.

Figure 1. Visualizing both project and system modularity.

The left-hand drawing shows the item defining process greet importing the definition of service console. On the right, the process is shown running as a client of that service.

Braces (curly brackets) denote the boundary of block scope, not sequential construction, as in C or Java. They may be omitted where no context is given, and thus no indication of scope required.

A process may be defined inline or offline in Honeysuckle with identical semantics. When defined inline, any further (offline) definitions must be imported above the description of the parent process.

```
...
{
  interface
    client of console
  defines
    String greeting : "Hello world!\n"

  send greeting to console
}
...
```

An inline definition is achieved simultaneously with command issue (*greet!*).

A process thus defined can still be named, facilitating recursion. For example, a procedure to create a new document in, say, a word processor might include the means by which a user can create a further document:

```
...
process new_document :
{
  ... context

  ...
    ...
       ...
       new_document
}
...
```

2.2. Simple Services

If all the console service does is eat strings it is sent, it could be very simply defined:

```
definition of service console

  imports
    object class String from StandardTypes

  service console :
  receive String
```

This is the sort of thing a channel can do — simply define the type of value that can be transmitted. Any such simple protocol can be achieved using a single service primitive. This is termed a *simple service*. Note that it is expressed from the provider perspective. The client must *send* a string.

One further definition is imported, of a string data type from a standard library — part of the *program environment*. It was not necessary for the definition of process greet to directly import that of String. Definitions in Honeysuckle are *transparent*. Since that of greet can see that of console, it can also see that of String. For this reason, no standard data type need be imported to an application program.

If more than one instance of a console service is required then one must define a *class* of service, perhaps called Console:

```
definition of service class Console
  ...
```

It is often very useful to communicate a “null datum” — a *signal*:

```
definition of service class Sentinel

  service class Sentinel :
  send signal
```

This example makes an important point. A service definition says nothing about *when* the signal is sent. That will depend on that of the process that provides it. Any service simply acts as a template governing the communication undertaken between two (or more) processes.

Signal protocol illustrates a second point, also of some importance. The rules governing the behaviour of every *service network component* (SNC) [3] do not require any service to necessarily become available immediately. This allows signal protocol to be used to *synchronize* two processes, where either may arrive first.

2.3. Service Construction and Context

Service protocol can provide a much richer interface, and thus tighter component specification, by constraining the order in which communications occur. Perhaps the simplest example is of *handshaking*, where a response is always made to any request:

```
definition of service class Console

  imports
    object class String from Standard_Types

  service class Console :
  sequence
    receive String
    send String
```

Any process implementing a *compound service*, like the above, is more tightly constrained than with a simple service.

A rather more sophisticated console might be subject to a small command set and would behave accordingly:

```
  service class Console :
  {
    defines
      Byte write : #01
      Byte read  : #02
    names
      Byte command

    sequence
      receive command
      if command
        write
          acquire String
        read
          sequence
            receive Cardinal
            transfer String
        ...
```

Now something strange has happened. A service has acquired *state*. While strange it may seem, there is no cause for alarm. Naming within a service is ignored within any process that implements it (either as client or provider). It simply allows identification between references within a service definition, and so allows a decision to be taken according the intended object or value. This leaves control over all naming with the definition of process context.

One peculiarity to watch out for is illustrated by the following:

```
service class Business :
{
  ...

  sequence
    acquire Order
    send Invoice
    if
      acquire Payment
        transfer Item
      otherwise
        skip
}
```

It might at first appear that payment will never be required and that service will always terminate after the dispatch of (a copy of) the invoice. Such is not the case. The above definition allows either payment to be acquired, then an item transferred, or no further transaction between client and provider. It simply endorses either as legitimate. Perhaps the business makes use of a timer service and decides according to elapsed time whether to accept or refuse payment if/when offered.

Although it makes sense, any such protocol is *not* legitimate because it does not conform to the formal conditions defining service protocol [3]. The sequence in which communications take place must be agreed between client and provider. Agreement can be made as late as desired but it must be made. Here, at the point of selection (`if`) there is no agreement. Selection and repetition must be undertaken according to *mutually recorded values*, which is why a service may require state.

A compound service may also be constructed via repetition. It might seem unnecessary, given that a service protocol is inherently repeatable anyway, but account must be taken of other associated structure. For example, the following might be a useful protocol for copying each week between two diaries:

```
service diary :
{
  ...

  sequence
    repeat
      for each WeekDay
        send day
    send week
}
```

It also serves as a nice illustration of the Honeysuckle use of an enumeration as both data type and range.

2.4. Implementation and Verification

Any service could be implemented in occam, using at most two channels — one in each direction of data flow. Like a channel, a service is implemented using rendezvous. Because, within a service, communications are undertaken strictly in sequence, only a single rendezvous is required. As with occam, the rendezvous must be initially empty and then occupied by the first party to become ready, which must render apparent the location of, or for, any message and then wait.

Each service can be verified via a finite-state automaton (FSA) augmented with a loop iteration counter. At process start, each service begins in an initial state and moves to its

successor every time a communication is encountered matching that expected. Upon process termination, each automaton must be in a final "accepting" state. A single state marks any repetition underway. Transition from that state awaits completion of the required number of iterations, which may depend upon a previous communication (within the same service). Selection is marked by multiple transitions leaving the state adopted on seeing the preceding communication. A separate state-chain follows each option.

Static verification can be complete *except* for repetition terminated according to state incorporated within the service. The compiler must take account of this and generate an appropriate warning. Partial verification is still possible at compile-time, though the final iteration count must be checked at run-time.

3. Shared and Distributed Services

3.1. Sharing

By definition, a service represents a contract between two parties only. However, the question of *which* two can be resolved dynamically. In the use of occam, it became apparent that a significant number of applications required the same superstructure, to allow services to be shared in this way.

occam 3 [20] sought to address both the need to establish a protocol governing more than one communication at a time and the need for shared access. *Remote call channels* effected a remote procedure call (RPC), and thus afforded a protocol specifying a list of parameters received by a subroutine, followed by a result returned. Once defined, RPCs could be shared in a simple and transparent manner. occam 3 also added shared groups of simple channels via yet another mechanism, somewhat less simple and transparent.

The RPC is less flexible than service protocol, which allows specifying communications in either direction in any order. Furthermore, multiple services may be *interleaved*; multiple calls to a remote procedure cannot, any more than they can to a local one. Lastly, the RPC is *added* to the existing channel abstraction of communication, complicating the model significantly. In Honeysuckle, services are all that is needed to abstract communication, all the way from the simplest to the most complex protocol.

Honeysuckle allows services to be shared by multiple clients at the point of declaration. No service need be explicitly designed for sharing or defined as shared.

```
{
  ...
  network
    shared console

  parallel
    {
      interface
        provider of console

      ...
    }
    ...  console clients
}
```

Any client of a shared service will be delayed while another is served. Multiple clients form an implicit queue.

3.2. Synchronized Sharing

Experience with occam and the success of *bulk-synchronous* parallel processing strongly suggest the need for barrier synchronisation. Honeysuckle obliges with the notion of *synchronized sharing*, where every client must consume the service before any can reinitiate consumption, and the cycle begin again.

```
...
network
  synchronized shared console
...
```

Like the sharing in occam 3, synchronized sharing in Honeysuckle is superstructure. It could be implemented directly via the use of an additional co-ordinating process but is believed useful and intuitive enough to warrant its own syntax. The degree of system abstraction possible is thus raised.

3.3. Distribution

Sharing provides a many-to-one configuration between clients and a single provider. It is also possible, in Honeysuckle, to describe both one-to-many and many-to-many configurations.

A service is said to be *distributed* when it is provided by more than one process.

```
...
network
  distributed validation
...
```

Note that the service thus described may remain unique and should be defined accordingly. Definition of an entire *class* of service is not required. (By now, the convention may be apparent whereby a lower-case initial indicates uniqueness and an upper-case one a class, with regard to any item — object, process, or service.)

The utility of this is to simplify the design of many systems and reduce the code required for their implementation. Again, the degree of system abstraction possible is raised.

A many-to-many configuration may be expressed by combining two qualifiers:

```
...
network
  distributed shared validation
...
```

When distributed, a shared service cannot be synchronized. This would make no sense, as providers possess no intrinsic way of knowing when a cycle of service, around all clients, is complete.

3.4. Design and Implementation

Neither sharing nor distribution influence the abstract interface of a component. Consideration is only necessary when combining components. For example, the designer may choose to replicate a number of components, each of which provides service A and declare provision distributed between them. Similarly, they may choose a component providing service B and declare provision shared between a number of clients.

A shared service requires little more in implementation than an unshared one. Two rendezvous (locations) are required. One is used to synchronize access to the service and the other each communication within it. Any client finding the provider both free and ready (both rendezvous occupied) may simply proceed and complete the initial communication. After this, it must clear both rendezvous. It may subsequently ignore the service rendezvous until

completion. Any other client arriving while service is in progress will find the provider unready (service rendezvous empty). It then joins a queue, at the head of which is the service rendezvous. The maximum length of the queue is just the total number of clients, defined at compile-time.

Synchronized sharing requires a secondary queue from which elements are prevented from joining the primary one until a cycle is complete. A shared distributed service requires multiple primary queues. The physical interface that implements sharing and shared distribution is thus a small process, encapsulating one or more queues.

4. Conclusion

Honeysuckle affords powerful and fully component-wise compositional system design and programming, yet with a simple and intuitive model for abstraction. It inherits and continues the simplicity of occam but has added the ability to express the component (or system) interface in much greater detail, so that integration and substitution should be more easily achieved. Support is also included for distributed and bulk-synchronous application design, with mobile objects and synchronized sharing of services.

Service (client-server) architecture is proving extremely popular in the design of distributed applications but is currently lacking an established formal basis, simple consistent model for abstraction, and programming language. Honeysuckle and PSA would seem timely and well-placed. Though no formal semantics for prioritisation yet appears to have gained both stability and wide acceptance, this looks set to change [21].

A complete programming language manual is in preparation, as is a working compiler. These will be completed and published as soon as possible.

Acknowledgements

The author is grateful for enlightening conversation with Peter Welch, Jeremy Martin, Sharon Curtis, and David Lightfoot. He is particularly grateful to Jeremy Martin, whose earlier work formed the foundation for the Honeysuckle project. That, in turn, was strongly reliant on deadlock analysis by, and the failure-divergence-refinement (FDR) model of, Bill Roscoe, Steve Brookes, and Tony Hoare.

References

[1] Ian R. East. The Honeysuckle programming language: An overview. *IEE Software*, 150(2):95–107, 2003.
[2] Per Brinch Hansen. *Operating System Principles*. Automatic Computation. Prentice Hall, 1973.
[3] 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.
[4] Inmos. *occam 2 Reference Manual*. Series in Computer Science. Prentice Hall International, 1988.
[5] 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.
[6] Fred R. M. Barnes and Peter H. Welch. Communicating mobile processes. In I. R. East and J. M. R. Martin et al., editors, *Communicating Process Architectures 2004*, pages 201–218. IOS Press, 2004.
[7] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall International, 1985.
[8] A. W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice-Hall, 1998.
[9] Peter H. Welch. Emulating digital logic using transputer networks. In *Parallel Architectures and Languages – Europe*, volume 258 of *LNCS*, pages 357–373. Springer Verlag, 1987.

[10] Peter H. Welch, G. Justo, and Colin Willcock. High-level paradigms for deadlock-free high performance systems. In R. Grebe et al., editor, *Transputer Applications and Systems '93*, pages 981–1004. IOS Press, 1993.
[11] 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.
[12] Jeremy M. R. Martin and Peter H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(3):1–18, 1997.
[13] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley, second edition, 2002.
[14] R. W. Floyd. Assigning meanings to programs. In *American Mathematical Society Symp. in Applied Mathematics*, volume 19, pages 19–31, 1967.
[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
[16] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
[17] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, ISE Inc., 270, Storke Road, Suite 7, Santa Barbara, CA 93117 USA, 1987.
[18] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
[19] Marcel Boosten. Formal contracts: Enabling component composition. In J. F. Broenink and G. H. Hilderink, editors, *Proceedings of Communicating Process Architecture 2003*, pages 185–197, University of Twente, Netherlands, 2003. IOS Press.
[20] Geoff Barrett. *occam 3 Reference Manual*. Inmos Ltd., 1992.
[21] Adrian E. Lawrence. Triples. In I. R. East and J. M. R. Martin et al., editors, *Proceedings of Communicating Process Architectures 2004*, Series in Concurrent Systems Engineering, pages 157–184. IOS Press, 2004.