*Communicating Process Architectures 2007*
*Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch*
*IOS Press, 2007*



# Concurrent/Reactive System Design with Honeysuckle

Ian EAST

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

ireast@brookes.ac.uk

**Abstract.** Honeysuckle is a language in which to describe systems with *prioritized service architecture* (PSA), whereby processes communicate values and (mobile) objects deadlock-free under client-server protocol. A novel syntax for the description of service (rather than process) composition is presented and the relation to implementation discussed. In particular, the proper separation of design and implementation becomes possible, allowing independent abstraction and verification.

**Keywords.** Client-server protocol, compositionality, component-based software development, deadlock-freedom, programming language, correctness-by-design.

## Introduction

Honeysuckle [1] is intended as a tool for the development of systems that are both concurrent and reactive (event-driven). Formal design rules govern the interconnection of components and remove the possibility of deadlock [2,3].

A model for abstraction is provided that is derived from *communicating process architecture* (CPA) [4]. Processes encapsulate information and communicate with each other synchronously. In place of the occam channel, processes send values or transfer objects to each other according to a *service* ("client/server" or "master-servant") protocol. Whereas a channel merely prescribes data type and orientation of data flow for a single communication, a service governs a series of communications and the order in which they can occur. It therefore provides for a much richer component interface [5].

In addition to describing service architecture, Honeysuckle also provides for the expression of reactive systems. A *prioritized alternation* construct [6] affords pre-emption of one process by another, allowing multiple services to interleave, while retaining *a priori* deadlock-freedom [3]. This allows the expression of systems with *prioritised service architecture* (PSA). One additional benefit of including alternation is that it overcomes the limitation of straightforward service architecture to hierarchical structure.

Honeysuckle also addresses certain short-comings of occam. It is possible to securely transfer objects between processes, rather than just copy values[1]. Provision is included for the expression of abstract data types (ADTs), and project-, as well as system-, modularity. Definitions of processes, services, and object classes, related by application, can be gathered together in a *collection.*

Previous papers have been concerned with the programming language and its formal foundation. This one is about Honeysuckle's support for proper engineering practice; in particular, how a PSA *design* may be expressed (and verified), independent of, but binding upon, any implementation. It is simple, yet powerful.

[1]Mobility has also been added in occam-$\pi$ [7].

## 1. The Problem of Engineering Software

### *1.1. Engineering in General*

In general, the term 'engineering' has come to mean a logical progression from specification through design to implementation, with each phase rendered both concrete and binding on the next. All successful branches of the discipline have found it necessary to proceed from a formal foundation in order to express the outcome of each phase with sufficient precision. Rarely, however, do engineers refer to that foundation. More common, and much more productive, is reliance upon design rules that embody necessary principles.

A common criticism of software engineering is that theory and practice are divorced. All too often, verification (of a design against specification) is applied *a posteriori*. This amounts to "trial and error" rather than engineering, and is inefficient, to say the least. Furthermore, verification typically requires formal analysis that is specific to each individual system. It requires personnel skilled in both programming and mathematics. In systems of significant scale, analysis is usually difficult and thus both expensive and error-prone.

The primary motivation behind Honeysuckle is to encapsulate analysis within the model for abstraction offered by a programming language. Adherence to formal design rules, proven *a priori* to guarantee security against serious errors, can be verified automatically at design-time ("static verification"). Both the cost and risk of error incurred by system-specific analysis can be thus avoided. "Trial and error" gives way to true engineering.

In order to serve as an engineering tool, Honeysuckle must fulfill a number of criteria.

### *1.2. Compositionality and the Component Interface*

Design is a matter of finding an appropriate component composition (when proceeding "bottom-up") or decomposition (when proceeding "top-down"). In order to 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

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*, that possesses the same interface, without affecting either the design or its compliance with a specification.

Software engineering now aspires to these principles [8].

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

With regard to the third criterion, clearly, listing a series of procedures, with given parameters, or a series of channels, with their associated data types, does little to describe object or process as a component. To substitute one object (process) with another that simply sports the same procedures (channels) would obviously be asking for trouble. One way of improving the situation is to introduce a finite-state automaton (FSA) between objects (processes) to govern the order of procedure invocation (channel communication) and thus constrain the interface [9]. Such a constraint is often termed a *contract*. The notion of a service provides an intuitive abstraction of such a contract, and is implemented using a FSA [5].

Honeysuckle is thus able at least to reduce the amount of ancillary logic necessary to adequately define a component, if not eliminate it altogether.

### *1.3. Balanced Abstraction*

It has long been understood that system abstraction requires an appropriate balance between data and control (object and process). This was reflected in the title of an important early text on programming — *Algorithms + Data Structures = Programs* [10]. Some systems were more demanding in the design of their control structure, and others in their data structure. An equal ability to abstract either was expected in a programming language.

Imperative programming languages emerging over the three decades since publication of Wirth's book have typically emphasized "object-oriented", while occam promoted "process-oriented", programming. While either objects or processes alone can deliver both encapsulation and a "message-passing" architecture, Honeysuckle offers designers the liberty to determine an appropriate balance in their system abstraction. This is intended to ease design, aid its transparency, and increase the potential for component reuse.

A programming language can obscure and betray abstraction. Locke showed how encapsulation, and any apparent hierarchical decomposition, can dissolve with the uncontrolled aliasing accepted in conventional "object-oriented" programming languages [11]. He also illustrated how the 'has' relation between two objects can become subject to inversion, allowing each to 'own' the other. State-update can be rendered obscure in a manner very similar to *interference* between two parallel or alternating processes.

Clearly, if modularity and transparency can break down even in simple sequential designs then it hardly bodes well for any extension of the model to include concurrency and alternation. The possibility then of multiple threads of control passing through any single object poses a serious threat to transparency and exponentially increases opportunity for error.

Honeysuckle applies strict rules upon objects: each object has but a single owner at any time, class structure is statically determined, and no reference is allowed between objects. All interaction is made manifest in their class definition, rendering interdependence explicit.

### *1.4. Separation of Design from Implementation*

Electronic engineering typically proceeds with the graphical capture of a design as a parallel composition of components interconnected by communication channels, collectively governed by precisely-defined protocol. This provides for both intuition and a precise concrete outcome. Modularity and compositionality impart a high degree of scalability.

One important principle at work is the clear separation of design and implementation. This has allowed electronic design to remain reasonably stable while implementation has moved from discrete devices, wires, and soldering irons, to VLSI and the FPGA.

All this remains an aspiration for software engineering.

This paper reports how Honeysuckle facilitates the separation of design from implementation. A sub-language expresses the behaviour of component or system purely in terms of communication. Design may be thus delivered: concrete, verified, and binding.

## 2. Process (De)Composition

### *2.1. Direct (One-to-One) Connection*

The simplest protocol between two processes may be expressed as a *simple service* [5]. A simple service is one comprising a single communication. It is equivalent to channel abstraction, where only data type and orientation of data flow is stipulated. As a result, anything that can be expressed using general communicating process architecture (CPA) and occam can be expressed using service architecture and Honeysuckle, but for a single constraint. There must be *no circuit* in the digraph that describes the system.

Since circuits can give rise to the possibility of deadlock, this is not a severe limitation. It does, however, remove the option to employ a very useful alternative design pattern for the proven denial of deadlock — *cyclic ordered processes* (COPs) [12,13,4]. The theoretical foundation for design rules that deny deadlock [14,15] allows for the composition of components, each guaranteed deadlock-free by adherence to a different rule. An appealing extension to Honeysuckle would be to allow the inclusion of ('systolic') COP arrays.

Service architecture, and especially *prioritised* service architecture, affords a much richer interface than channels allow. Much more information can be captured in a design. The behaviour of system or component can be expressed in terms of communication protocol alone, without reference to procedure.

It may seem odd to define a system with reference only to communication and not to 'physical' entities like objects or processes. But a system can be very well described according to the way it communicates. It can, in fact, be defined this way. An emphasis on communication in a specification often leads to concurrency and alternation in implementation. It is only natural to retain such emphasis within a design. Honeysuckle offers a simple way to abstract such behaviour to a degree intermediate between specification and implementation, and in a manner open to intuitive graphical visualization.

For example, suppose a system is built around a component that offers a single service, which is *dependent* upon the consumption of just one other (Figure 1).


**Figure 1.** A single service dependent on just one other.

We can express this simply:

```
network
  s1 > s2
```

Note that the symbol used to denote a dependency is suitably asymmetric, and one that also correctly suggests the formation of a partial order.

As it stands, the above forms a complete system, implemented as a parallel composition. The centre component, isolated, requires an INTERFACE declaration:

```
interface
  provider of s1
  client of s2
```

but no network definition. A complete system requires no interface declaration.

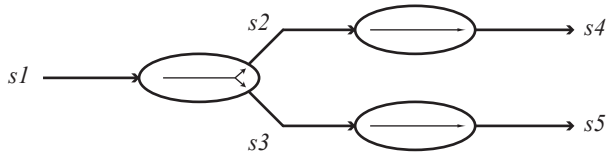


**Figure 2.** A tree structure for service dependency.

A tree-structured component (Figure 2) is easily described:

```
network
  s1 > s2, s3
  s2 > s4
  s3 > s5
```

Chains of identical services can be indicated via replication of a dependency:

```
network
  repeat for 2
    s1 > s1
```

Note that all reference to services has been solely according to their type. No instance of any service has yet needed distinction by name. Honeysuckle can connect processes correctly simply by their *port*[2] declarations and the network definition that governs their composition.

Naming might have become necessary should one component provide multiple identical services, except that such structure may be described more simply.

### *2.2. Sharing and Distribution*

A common CPA design pattern is the consumption of a common service by multiple clients, which is why occam 3 introduced *shared channels* [16]. Honeysuckle similarly permits the sharing of any service. For example, Figure 3 depicts a simple closed system where two components share a service.

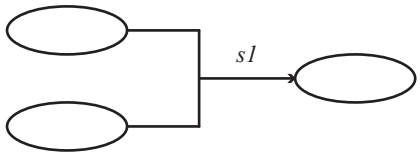


**Figure 3.** Sharing of a service between two clients.

Such a design is declared as follows:

```
network
  shared s1
```

As outlined in a previous paper [5], one-to-any, and any-to-any connection patterns are also supported via the DISTRIBUTED and SHARED DISTRIBUTED attributes, respectively. None of these options are the concern of implementation. They neither appear in nor have any effect upon the interface of any single component.

Within a design, there is still no need for naming instances of service. We have thus far presumed that every service is provided in precisely the same way, according to its definition only, and subject to the same dependencies.

### *2.3. Service Bundles, Mutual Exclusion, and Dependency*

A design may require a certain *bunch* of services to be subject to mutual exclusion. If any member of the bunch is initiated then all of the others become unavailable until it completes.

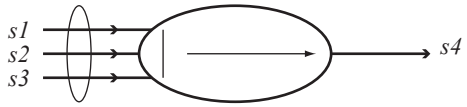


**Figure 4.** A service *bunch*, subject to mutual exclusion and a dependency.

Connections to the component depicted in Figure 4 can be expressed:

[2] A *port* is one end of a service, *i.e.* either a *client* or *server* connection.

```
network
  exclusive
    s1
    s2 > s4
    s3
```

A bunch of mutually exclusive services can be provided by a single, purely sequential, process. All that is required is selection between the initial communications of each. In occam, an ALT construct would be employed. The body of each clause merely continues the provision of the chosen service until completion. An outer loop would then re-establish availability of the entire bundle. (In PSA, and in CPA in general, it is often assumed that processes run forever, without terminating.)

An object class, when considered as a system component, typically documents only procedures offered, within its interface. It does not usually declare other objects on which it depends. A service network component (SNC) documents both services provided *and* services consumed, together with the dependency between. Any interface thus has two 'sides', corresponding to provision and consumption, respectively. Honeysuckle requires documentation of dependency beginning with service provision and progressing towards consumption.

Suppose a system including the component shown in Figure 4 were to be extended with $s4$ being provided under mutual exclusion with another service, $s5$, and a dependency upon the consumption of yet another, $s6$. We would then write:

```
network
  exclusive
    s1
    s2 > s4
    s3
  exclusive
    s4 > s6
    s5
```

If $s4$ failed to reappear under the second EXCLUSIVE heading, $s5$ (and any other services in that bundle) would be listed together with $s1 - 3$. Mutual exclusion is fully associative.

### *2.4. Service Interleaving*

An alternative to bunching services under mutual exclusion is to allow them to *interleave*. This allows more than one service in a group to progress together. Should two be ready to proceed at the same moment, the ensuing communication is decided according to service[3] *prioritization*. A service of higher priority will pre-empt one attributed lower priority.

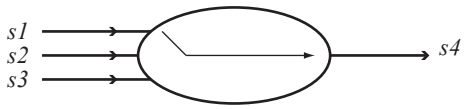


**Figure 5.** Interleaving services.

This too can be expressed as a feature of design, quite separate from implementation:

```
network
  interleave
    s1 > s4
    s2
    s3
```

[3]Each member of any bunch is attributed a common priority.

Prioritisation is indicated simply by the order in which services are listed (highest uppermost in both picture and text).

A process might interleave bunches. Each bunch would remain subject to mutual exclusion between its members:

```
network
  interleave
    exclusive
      s1 > s4
      s2
      s3
    exclusive
      s5
      s6
      s7 > s8
```

Again, implementation reduces to a programming construct; in this case, *prioritized alternation* (WHEN) [6]. Each clause in a Honeysuckle alternation may be a guarded process or a selection, according to whether a single service or a bunch is offered.

Interleaving several instances of a common service offers an alternative to sharing a single instance, where each client is effectively allocated the same priority. Replication may be used to indicate vertical repetitive structure, as it can horizontal:

```
network
  interleave for 2
    exclusive
      s1
      s2 > s4
      s3
```

Note that replication under mutual exclusion would add nothing to the notion of sharing.

## 3. Asymmetry in Service Provision

For many systems with PSA, it is enough to define their design without distinction between two instances of the same service type. Implementation could proceed with components whose interface can be defined with reference only to that type. If two different processes each declare the capability of providing that type of service, it would not matter which provides each instance of it.

Any departure from that scenario is termed an *asymmetry*. There are two kinds.

A *design asymmetry* is one where dependency in the provision of two services of the same type differs. An example might be formed were $s3$ in Fig. 2 replaced by a second use of $s2$. This would make it impossible to document dependency without ambiguity. Note that no such ambiguity would result upon implementation since component interface could be matched with dependency. A (reasonably intelligent) compiler will still be able to compose components correctly.

Note that any service is necessarily shared (or distributed) symmetrically, since no provider (or client) can distinguish one client (provider) from another.

An *implementation asymmetry* is where the provision of two instances of the same service are not interchangeable, even though there may be no design asymmetry. Some relationship between information exchanged is material to the system required. If so then a single instance may neither be shared nor distributed.

It is worth reflecting that, in traditional, typically purely sequential, programming, we commonly distinguish between “data-oriented” and “control-oriented” application design. Often, the orientation is inherent in the problem. Sometimes, it is a choice reflecting that

of the designer. One might similarly identify "service-orientation" also. Business nature and organization has re-oriented itself towards service provision, to great effect. The same development in the design of software would arguably result in a greater reliance upon service architecture, with less asymmetry appearing.

At the cost of complicating the declaration of design a little, a mechanism is provided by Honeysuckle by which asymmetry may be introduced. For each asymmetric use of a service, a *service alias* ('renaming') is declared within the network declaration. It then becomes possible for the interface declaration of each process to distinguish one instance of service from another of the same type.

If we again refer back to Fig. 2 for an example, let us suppose that $s2$ and $s3$ are of the same type (share the same definition), and $s4$ and $s5$ are similarly alike. Suppose that we *care* that each instance of $s2$/$s3$ is provided separately, because there is some difference we cannot yet make explicit. All we need do is declare two service aliases within the network definition:

```
network
  named
    s2 : s3
    s4 : s5
  ...
```

Each component interface can now distinguish the desired connection.

## 4. Parametric and Dynamic Configuration

Modular software engineering calls for the ability to compose components whose utility is *not* restricted to a single application. Having renamed services in order to apply an implementation asymmetry in service provision, it should be possible to employ a component designed for wider use. While it must possess an interface appropriate to any design asymmetry, it will know nothing of any service alias. Its interface will refer only to the original service (type) names given in each corresponding definition.

An in-line component definition can match service and alias directly:

```
{
  ...
  network
    named
      s2 : s3
      ...

  parallel
    {
      interface
        provider of s2 alias s3

      ...
    }
    ...
}
```

while the option remains to state simply "`provider of s3`".

The interface of any 'off-line' process definition can indicate that it expects to be told which service it is to consume/provide via `alias ?`, in which case its reference (invocation) should provide a *configuration parameter*.

There is one other kind of configuration parameter, used by the network declaration of the recipient. A *configuration value* may be passed and used to limit replication. Since

this may be computed upon passing, it allows the network of a parallel component to be configured dynamically.

A Honeysuckle process reference may thus include up to three distinct actual parameter lists, arranged vertically ("loo roll" style), and delimited by semi-colon. When each list has no more than one item, parameters can be arranged on the same line as the command (process invocation). For example, suppose a process *mediate* is defined separately (like a procedure in Pascal), and it expects one service alias and one configuration value. Definition would be as follows:

```
process mediate is
  {
    ...
    interface
      client of s1 alias ?
      ...

    network
      received Length
      interleave for Length
        ...
  }
```

An invocation might be simply:

```
mediate ; s2 ; 4
```

## 5. Conclusion

Honeysuckle began as a single-step method for the composition of concurrent/reactive software guaranteed free from the threat of deadlock. As such, it was either going to remain a simple academic exemplar, or grow into a tool suited to professional use. It was decided to take the latter path, which has inevitably proved long and arduous.

Here, elements of the language have been introduced that afford PSA *design*, separate from, and independent of, implementation. Design of system or component is expressed purely in terms of communication, as a composition of services rendered. Any such design may be compiled and verified independently, and automatically, using the same tool used for implementation. It will then remain binding as the implementation is introduced and refined. Every verified design, and thus implementation, is *a priori* guaranteed deadlock-free.

It has been shown how a design may be composed under service dependency, mutual exclusion, and interleaving, and how repetitive structure can be efficiently expressed. While prioritized service architecture alone may suffice to abstract some systems, especially when design is oriented that way, others may call for significant emphasis on process rather than communication. A mechanism has therefore been included whereby *asymmetry* in service implementation can be introduced.

Given that the parallel interface of each component is defined purely according to services provided and consumed, *configuration parameters* have proved necessary in order to allow the reuse of common components, and preserve modularity. They also afford limited dynamic configuration of components, allowing the structure of each invocation to vary.

With regard to the progress of the Honeysuckle project, another decision taken has been to complete a draft language manual before attempting to construct a compiler. A *publication language* would then be ready earlier, to permit experiment and debate. This is now complete, though the language (and thus manual) is expected to remain fluid for some time yet [17].

Work is now underway towards a compiler. A degree of platform independence will be facilitated by the use of *extended transputer code* (ETC) [18] as an intermediary[4].

While Honeysuckle has evolved into a rather ambitious project, it is nonetheless timely. The beginning of the twenty-first century has marked the rise of large embedded applications, that are both concurrent and reactive. Consumers demand very high integrity from both home and portable devices that command prices, and thus (ultimately) development costs, orders of magnitude below those of traditionally challenging applications, such as aerospace. Existing methods are inappropriate. While a sound formal foundation is an essential prerequisite for something new, proper support for sound engineering practice is also required.

Honeysuckle now offers both.

By clearly separating design from implementation, while rendering it inescapably formal and binding, Honeysuckle brings the engineering of software into closer harmony with that of electronic and mechanical systems, with which it must now co-exist.

## References

[1] Ian R. East. The Honeysuckle programming language: An overview. *IEE Software*, 150(2):95–107, 2003.
[2] 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.
[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] Ian R. East. *Parallel Processing with Communicating Process Architecture*. UCL Press, 1995.
[5] 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.
[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] 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.
[8] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley, second edition, 2002.
[9] 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.
[10] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Series in Automatic Computation. Prentice-Hall, 1976.
[11] Tom Locke. Towards a viable alternative to OO — extending the occam/CSP programming model. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, *Proceedings of Communicating Process Architectures 2001*, pages 329–349, University of Bristol, UK, 2001. IOS Press.
[12] E. W. Dijkstra and C. S. Scholten. A class of simple communication patterns. In *Selected Writings in Computing*, Texts and Monographs in Computer Science, pages 334–337. Springer-Verlag, 1982. EWD643.
[13] Jeremy Martin, Ian East, and Sabah Jassim. Design rules for deadlock freedom. *Transputer Communications*, 2(3):121–133, 1994.
[14] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. Technical Report PRG-57, Oxford University Computing Laboratory, 8-11, Keble Road, Oxford OX1 3QD, England, 1986.
[15] S. D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, 4:209–230, 1991.
[16] Geoff Barrett. *occam 3 Reference Manual*. Inmos Ltd., 1992.
[17] Ian R. East. *The Honeysuckle Programming Language: A Draft Manual*. 2007.
[18] Michael D. Poole. Extended transputer code — a target-independent representation of parallel programs. In P. H. Welch and A. W. P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 187–198. IOS Press, 1998.

[4]Subject to the kind permission of Prof. Peter Welch and his colleagues at the University of Kent