State-based Modelling of Functions and Pump Engines

Graham G. Thomason

Report Relating to the Thesis "The Design and Construction of a State Machine System that Handles Nondeterminism"

Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

July 2004

Summary

Many problems of integration testing can be addressed if adequate state-based models are available. This paper discusses how to model the queuing, dequeuing and dispatching of function-call messages as processed by a pump engine. A few extensions to the standard state-based paradigm are introduced from the domain of conventional imperative languages and techniques such as "C".

Contents

1. Introduction 1

2. Modelling function calls 3
   - 2.1 Synchronous function calls 3
   - 2.2 Asynchronous function calls 9
   - 2.3 Mixed synchronous / asynchronous calls 14
   - 2.4 Syntax issues for synchronous and asynchronous function calls 14

3. Parameter passing 15

4. Modelling a pump engine 20
   - 4.1 Introduction 20
   - 4.2 Unsuspendable producers 23
   - 4.3 Suspendable producers 27

5. Summary of syntax issues 33

6. References 35

1. Introduction

The emerging solution to the 'software crisis' – the runaway complexity, size and lead time of software systems – is *component-based synthesis* of systems. Microsoft has deployed COM for a number of years. Philips is applying the basic concept to embedded systems (e.g. Koala in CE, and a component-like API in DVP-2). ***Technically***, components provide for system modularity, customisability, maintainability and upgradeability. ***Commercially***, they provide a market in software modules based on open standards, but with protection of proprietary implementation details. For Philips Semiconductors, this gives opportunities to market re-usable software. For Consumer Electronics, this provides a way to build large systems by selecting and binding components from repositories, rather than by writing millions of lines of code.

The hard part is ***integration testing*** of systems built from components, typically from different suppliers (so with fair scope for disparate interpretations of the interfaces and other incompatibilities). Integration testing is essential to achieve a reliable system [Trew]. Since the primitive elements of components are *functions* (organized in *interfaces*), good modelling capabilities of functions are an important pre-requisite to adequate component modelling. Multi-threaded systems, such as television systems, (including the MG-R software architecture), typically need to queue function calls, and to do this they make extensive use of pump engines. Pump engines can be encapsulated in software components –with all the concomitant advantages and disadvantages of this upcoming technology.

The author's earlier report, [CompBinding], focused on component composition and introduced a concept for handling recursive functions. The present report builds on that concept with the aim of designing a developer-friendly way of specifying the state behaviour of synchronous and asynchronous functions which may be self- or mutually recursive, and which may be pumped. The issue of how the models can be implemented in a state machine engine such as STATECRUNCHER is also considered.

The motivation for work on state-based modelling, in particular function modelling, is the need for *automated integration testing* of systems built using component technology. Examples of component-based approaches within Philips are Koala [Koala, KoalaYP] in CE, and DVP-2 [vAntwerpen, Kunst] in Semiconductors.

Further, the report describes how a pump engine can be modelled, i.e. the queuing, dequeuing and dispatching of function calls.

STATECRUNCHER, a language and engine for modelling state behaviour, has been designed for integration testing of software components [StCrMain]. This report is written from a perspective of how the techniques presented might be realized in STATECRUNCHER (of which some knowledge is assumed).

The state diagram notation is as described in [StCrMain], which is largely compatible with UML notation. In places an abbreviated syntax will be used, (particularly for actions, which simply follow the symbol '/', rather than using surrounding braces and the keyword `fire` before events). Declarations of events and variables are indicated by the following symbols.

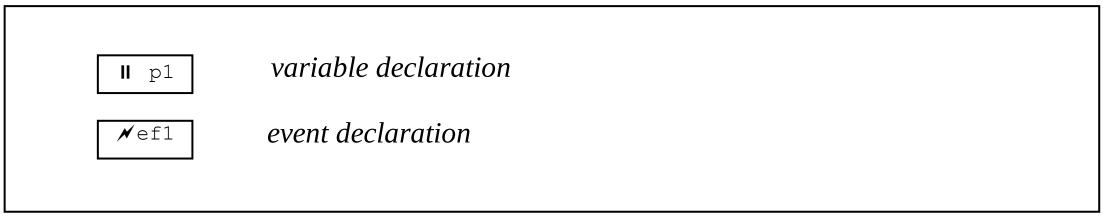


Figure 1. Symbols for variable and event declarations.

A typical component scenario for function call and parameterized event methods is as shown below:

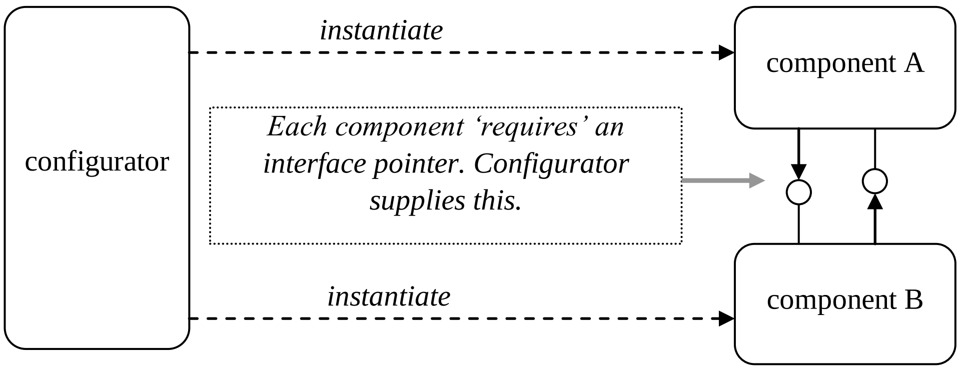

instantiate
component A
configurator
Each component 'requires' an interface pointer. Configurator supplies this.
instantiate
component B

Figure 2. Component-based system

Component A will call functions in component B and vice versa. But the models of A and B (for simulation and testing) will not know who their clients are. Components may contain pump engines and so support queuing of functions to be dispatched later.

2. Modelling function calls

2.1 Synchronous function calls

In a synchronous function call the function executes on the caller's thread. When the function returns, it is regarded as complete. Clearly, the caller cannot make a second function call until a synchronous call completes, since the thread of control is not available.

Within the category of synchronous function calls, we can distinguish between **bound** calls and **unbound** calls.

Bound calls

A bound call will run to completion without requiring any more events to drive it to completion. It is typically CPU-bound – an example is a function to find the maximum of a list of numbers. Alternatively it might involve some activity which frees up the CPU (e.g. by performing some I/O), but the execution is regarded as predetermined rather than dependent on the presence of an event. If the function call is not modelled as an atomic occurrence, i.e. if there can be an intervening event between start and completion, then the call is better modelled as an unbound one (see below).

A bound call can be modelled as a simple library function in an assignment action on a transition:

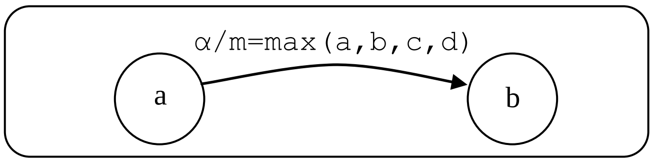


Figure 3. Bound call

A bound call (like other kinds of call) is capable of firing other events. A standard library module will provide for this.

Unbound calls

An unbound call requires the occurrence of at least one additional event to drive the function to completion. For example, if a function obtains input from a user, we might model it as requiring the event "input_obtained" to complete it.

We initially consider the easier case of restricting the user to one call per transition. We make single implantations per set member on the call. The figure below illustrates the process, with explanations following.

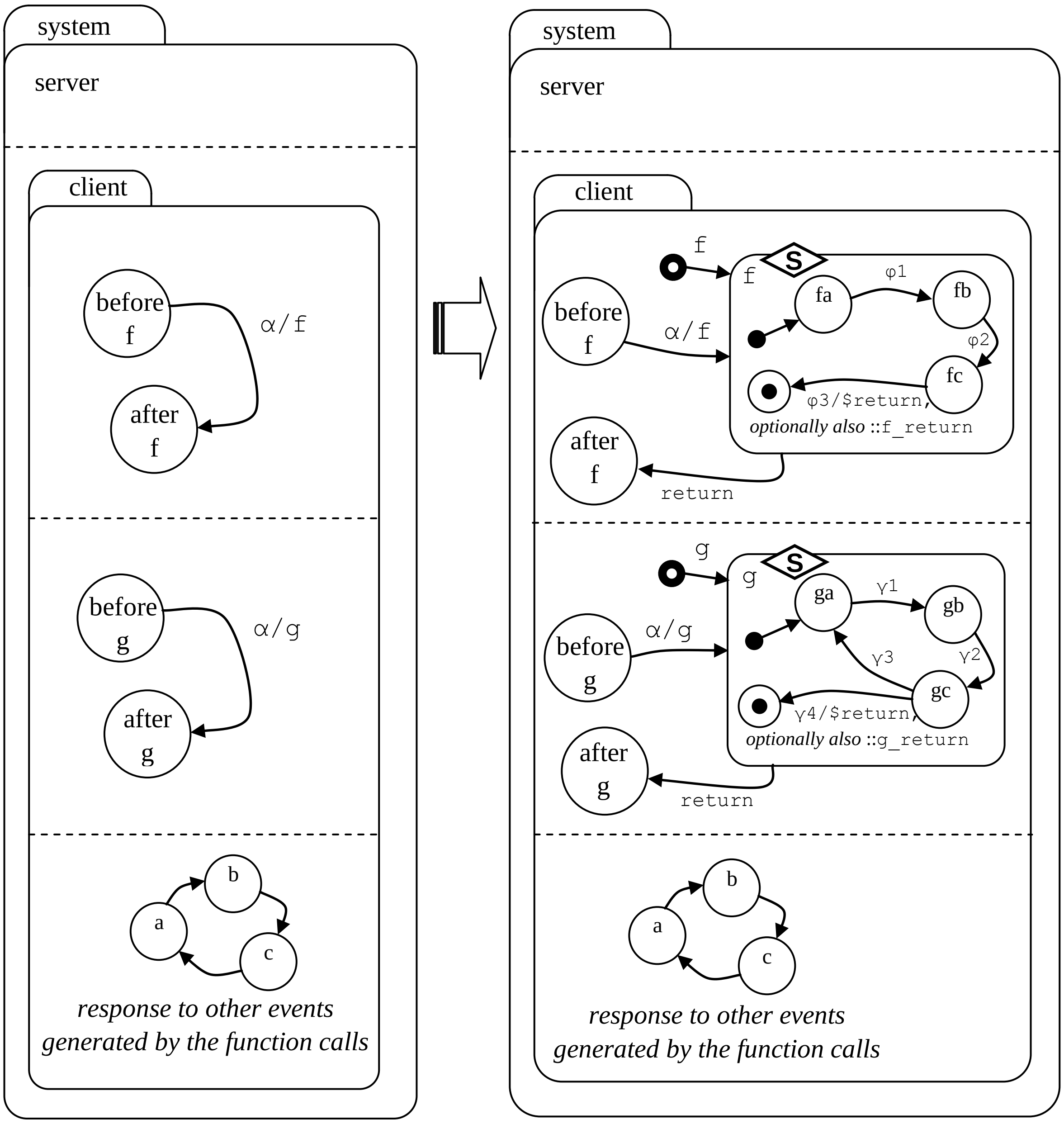
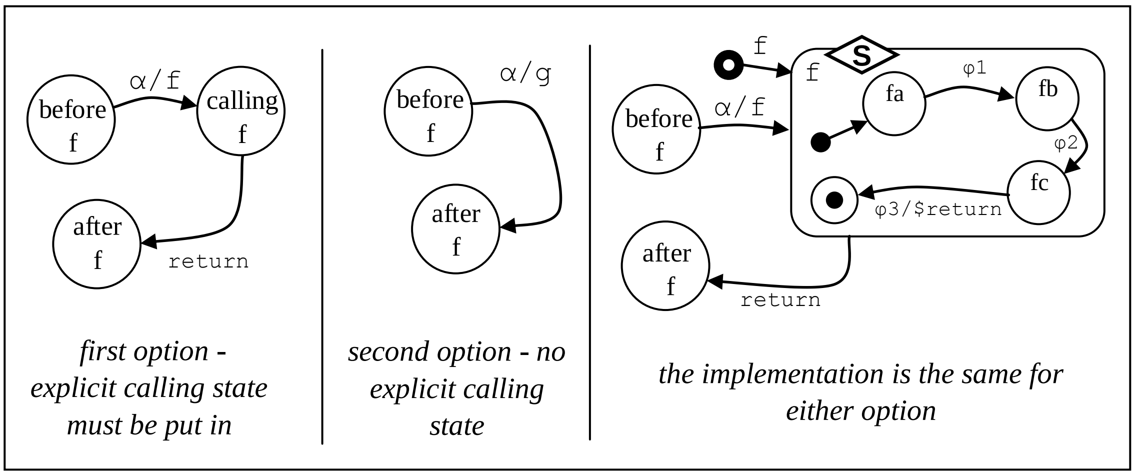


Figure 4. Implementation of unbound synchronous calls

Explanation and notes on synchronous calls:

1. The optional fired events on the right hand side of the above figure, marked *"optionally also* `::f_return`*"* and "*optionally also* `::g_return`*"* are included for completeness because they would be needed in the event of a pumped call. The "`::`" indicates global scope. Pumped calls are discussed in chapter 4. These optional events do not play a role in the present discussion.

2. Two options were considered for representing the client side call –one with an intermediate "calling" state and one without, illustrated below, but both corresponding to the same implementation in principle. The second is perhaps more developer friendly, and we adopt it in general in this report.



before f
α/f
calling f
after f
return
first option - explicit calling state must be put in
before f
α/g
after f
second option - no explicit calling state
f
S
f
before f
α/f
fa
φ1
fb
φ2
fc
φ3/$return
after f
return
the implementation is the same for either option

Figure 5. User options considered for synchronous calls

3. The arrow ⇨ indicates the dynamic transformation of the state machine when the event occurs. A new state machine element is implanted on the calling event - indicated by ⊙→ . It is not adequate to perform a static implantation of the function (e.g. at compile time) as some functions will be recursive; it is only at run-time that the termination condition can be identified and executed. Note that the tip of the transition is to the newly created element, (not the state indicated by the user, which will be reached on function completion).

4. The clusters marked **<S>** are implanted on call as mentioned and removed on completion. The **S** stands for **S**ynchronous, and is a variation of the **<R>** for **R**ecursive of [CompBinding].

5. The standard notation ●→ shows what state is entered on creation of the implantation.

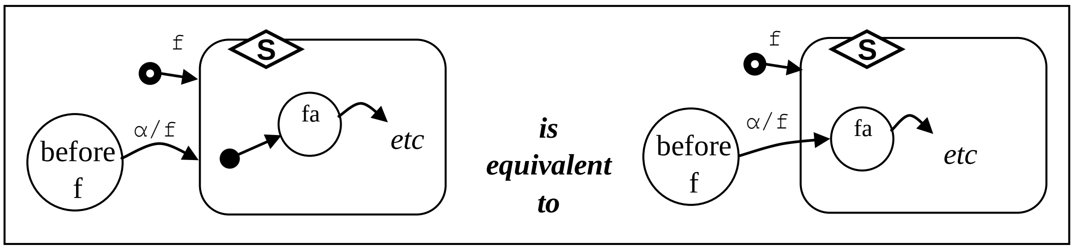


Figure 6. State entered on implantation creation

Functions do not remember state history from one invocation to another.

6. The notation ⦿ (which we call a *terminator*) indicates that the implantation is to be removed after a transition entering it. Suggested syntax for this symbol: the keyword `void.` There can be several separate transitions to the terminator.

7. Just as only one default state is allowed in conventional state machines, so there should only be one initial state in implantable machines.[1] If necessary, fork nondeterminism can be applied on the next transition.

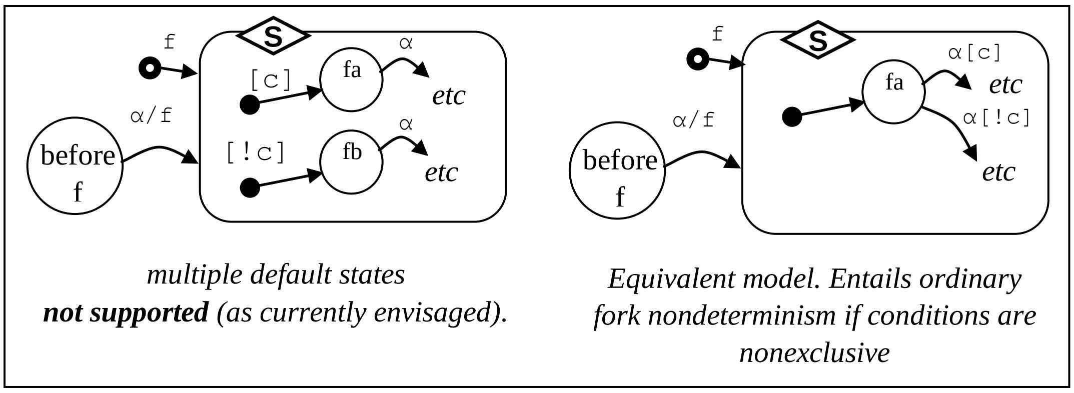

f
S
before f
α/f
[c]
fa
α
etc
[!c]
fb
α
etc
multiple default states
***not supported** (as currently envisaged).*
f
S
before f
α/f
fa
α[c]
etc
α[!c]
etc
Equivalent model. Entails ordinary fork nondeterminism if conditions are nonexclusive

Figure 7. Equivalent model for nondeterministic initial state

8. Although the implantations shown are clusters, they could be sets (parallel state machines), or possibly leaf-states.

[1] If multiple default states on nonexclusive entry conditions were supported, this would entail nondeterminism, "default state fork nondeterminism".

9. Whatever the option chosen for the *user representation*, the *implementation* could be make use of the technique used by asynchronous calls. This is described in detail in section 2.2; it amounts to the following in this case:

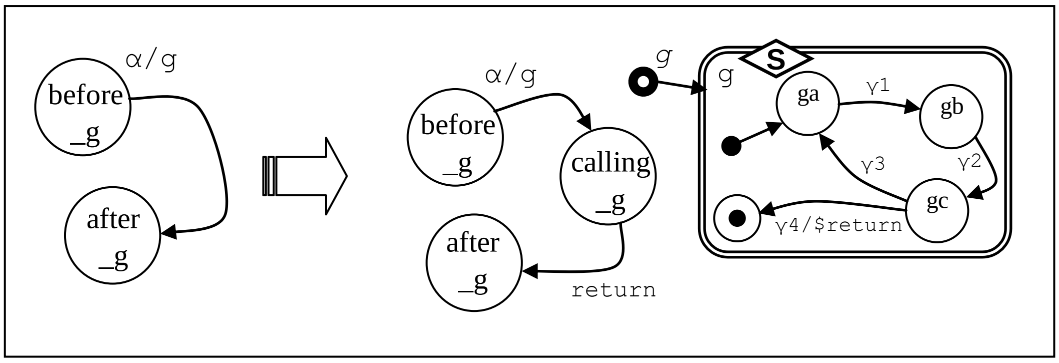


Figure 8. Synchronous implemented compatibly with asynchronous

The user cannot make use of any apparent asynchronous functionality here, because the state marked `calling_g` (or some system-generated name) and is not accessible for any other transitions. The client is locked in this state until the return event is fired.

Multiple synchronous function calls

If several synchronous calls are put on transition, they are necessarily to be interpreted as a sequence. The implementation can translate this into a chain of transitions, either built up in one go or created step by step. The principle is illustrated in the following figure.

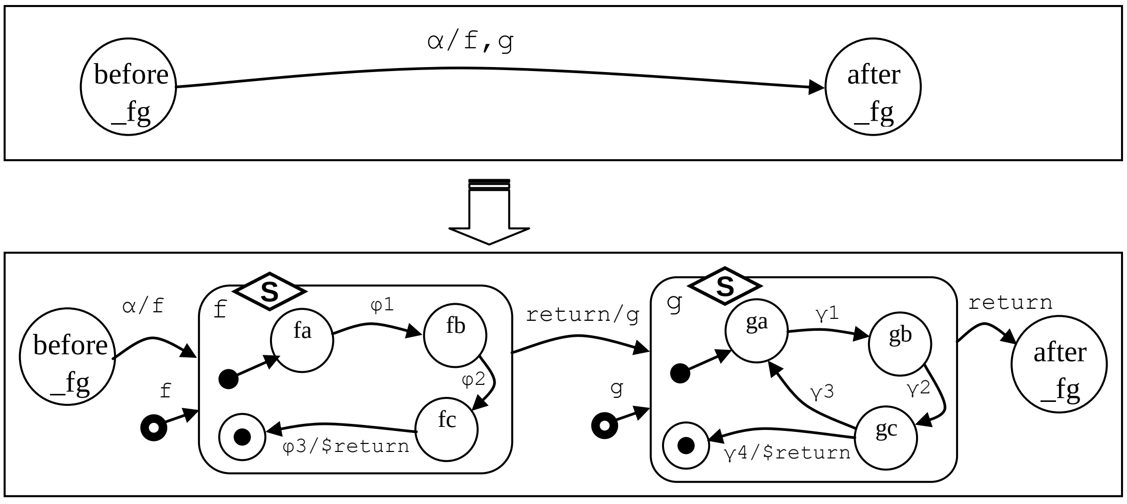


Figure 9. Multiple Sequential calls - automatic intermediate states (1)

If the sequence of actions contains a mixture of function calls and other actions, e.g.

`α/f, x=p+q, g,y=z, β, h , x=x+1`

where `f`, `g`, and `h` are function call events, but β is just a global event, then chaining must put the non-function-call actions on the linking transitions:

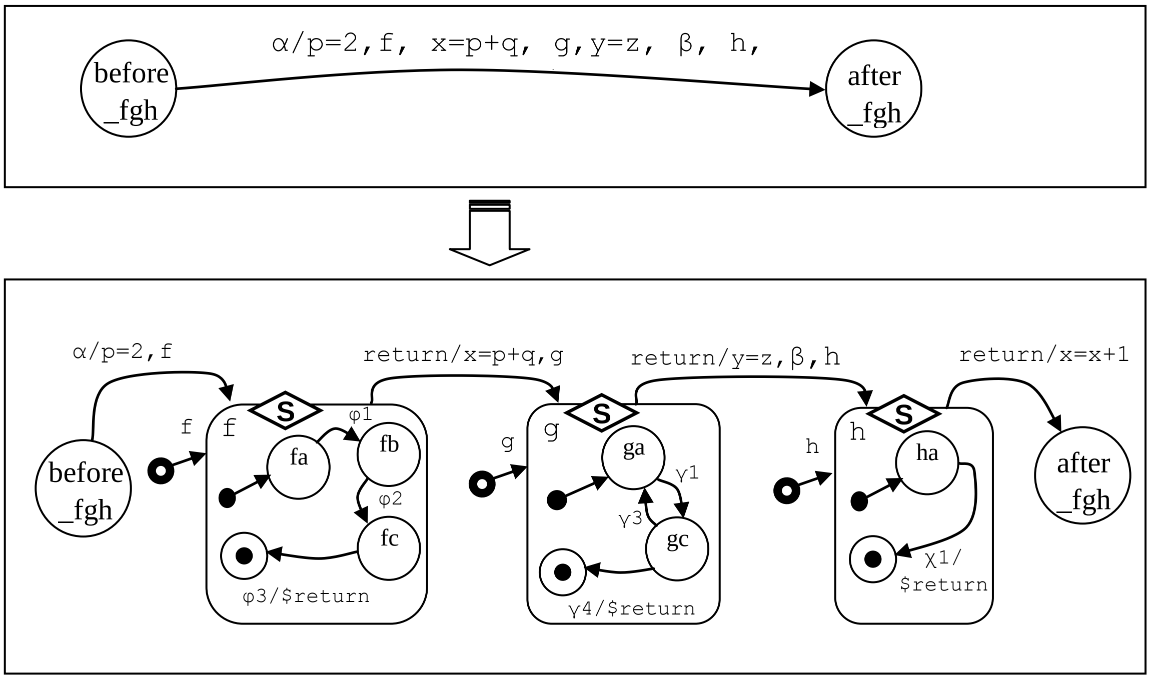


Figure 10. Multiple Sequential calls - automatic intermediate states (2)

The exact sequence (or: micro-step) in which the non-function-call actions, including the initial one, take place, is dependent on the precise transition algorithm. This is not discussed further in this report.

2.2 Asynchronous function calls

An asynchronous function call provides return of control to the caller, called a *pending* return, but the function will continue to do some (or all) serious processing on another thread. When the processing associated with the call is complete, the function provides a *notification*. It is possible that intermediate and final notifications are given, perhaps indicating phases of processing, but any final notification will mean that no more notifications can come from this function invocation. The intermediate notifications can just be regarded as ordinary broadcast events.

An asynchronous function may, depending on run-time circumstances, provide *either* a synchronous-like completion *or* a pending return with later notification. Take for example a request for a web page. If the page is in cache, it may be quickly returned with completion. Otherwise, the function will return pending, access the page over the Internet, and notify when it has been obtained.

It is possible to have several asynchronous function calls outstanding at any one time, each running on their own thread. For this reason, in a state model, the caller and all called functions must be able to transition independently. Also, we must have some way to distinguish pending, notify and other events; if they are simply called e.g. `notify`, there will be ambiguity as to which function produced it (unlike the synchronous function case). Solutions might be to parameterize the events or to use names related to the function name.

Implementation

When an event representing an asynchronous call is processed, a state-machine element is implanted (as in the unbound synchronous case), but with a special status indicated by a double perimetral line. The 'special status' ensures that the implantation has a specific *scope* that is effectively local to the caller, and a *lifetime* that is independent of that of caller. This is discussed further in the notes following the figure.

For multiple function calls, a temporary implantation of a machine element is performed per function called.

The next figure illustrates the principle, with explanations following.

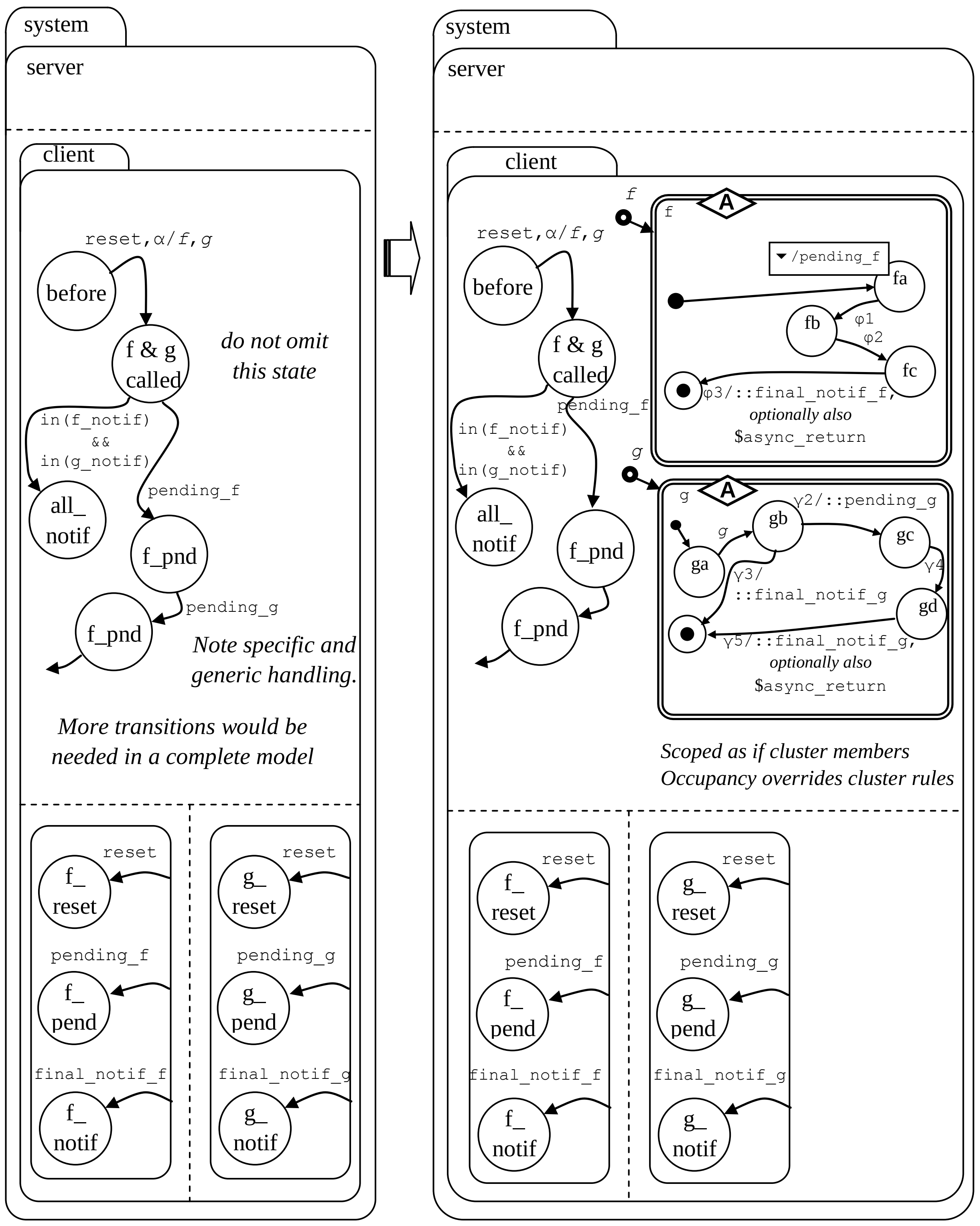

system
server
client
reset,α/f,g
before
f & g called
do not omit this state
in(f_notif) && in(g_notif)
all_ notif
pending_f
f_pnd
pending_g
Note specific and generic handling.
More transitions would be needed in a complete model
reset
f_ reset
g_ reset
pending_f
f_ pend
pending_g
g_ pend
final_notif_f
f_ notif
final_notif_g
g_ notif
f
A
/pending_f
fa
fb
fc
φ1
φ2
φ3/::final_notif_f, optionally also $async_return
g
ga
gb
gc
gd
γ2/::pending_g
γ3/ ::final_notif_g
γ4
γ5/::final_notif_g, optionally also $async_return
Scoped as if cluster members
Occupancy overrides cluster rules

Figure 11. Implementation of asynchronous calls

Explanations and notes on asynchronous calls:

1. Notification events are global (indicated by the "`::`" operator). This ensures that all recipients will see them, –but the name must be unique. The optional fired events on the right hand side of the above figure, marked *"optionally also* `$async_return`*"* are included for completeness because they would be needed in the event of a pumped call. Pumped calls are discussed in chapter 4. These optional events do not play a role in the present discussion.

2. Asynchronous functions need to be distinguished syntactically from synchronous calls, perhaps by the keywords `synchronous` and `asynchronous`.

3. The implanted machine elements, marked **<A>**, (for **A**synchronous) have the ***scope*** of being a sibling of the element at the tip of the transition arrow. The scope is *not* related to the effective source (leaf-)state, nor the orbital state (if present). This fixed policy should facilitate precise targeting of broadcast events and variables where scoping operators are used.

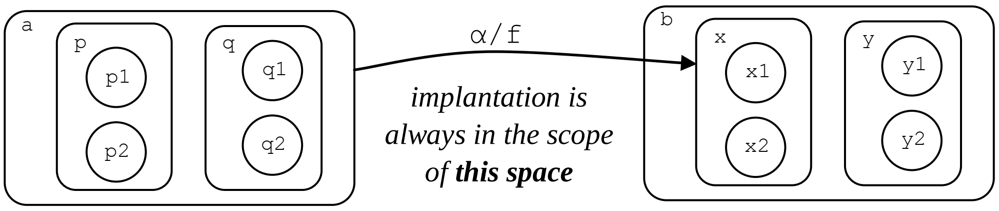


Figure 12. Scope of implantation

4. The implanted machine elements, marked **<A>**, and with a double perimetral line, could be regarded as ***extra active members***, of their parent. Thus in the case of the parent being a cluster, they ***break the ordinary rule*** that only one active member is allowed in a cluster. However, the implantation can perhaps better be thought of as rather independent of its machine-path parent, since it has an independent life-cycle. The parent may not have the implantation marked as a child, so that the implantation will not take part in algorithms which examine a parent's children. In this way, a cluster can be exited, for example, without interfering with the life-cycle of the implanted member.

Note that if we were to avoid breaking the ordinary rule, we might consider placing the implantation as a co-set member w.r.t. the caller, which would be one level too high, as it would not identify which co-set-member the caller was.

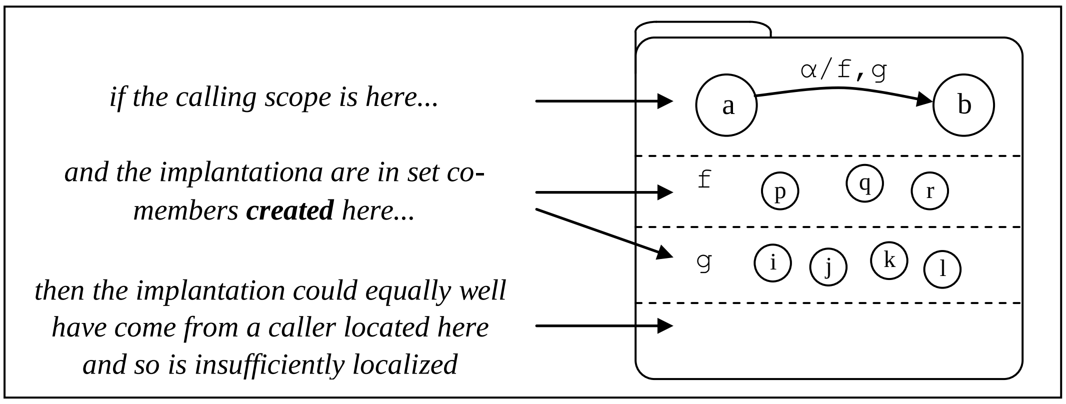

if the calling scope is here...
and the implantationa are in set co-members created here...
then the implantation could equally well have come from a caller located here and so is insufficiently localized
α/f,g
a
b
f
p
q
r
g
i
j
k
l

Figure 13. Implantation as a co-set-member is at a level too high

Another attempt to avoid breaking the cluster rule is to wrap the caller in a new set, with the function implantations as co-set-members. Although this preserves the cluster rule, it changes the scope of the caller, (making the machine path one level deeper), and so makes scoping and targeting the caller, (from anywhere else in the entire state machine) difficult. Therefore, this has to be rejected as well.

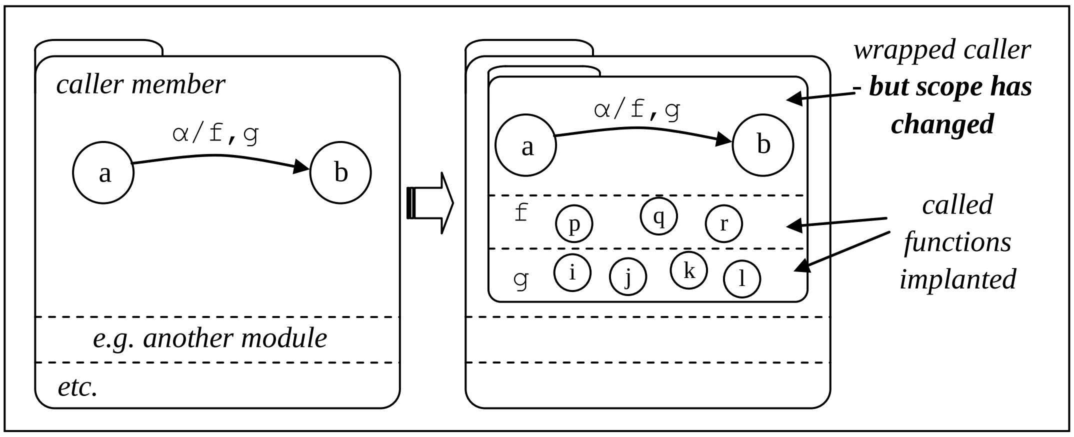

caller member
α/f,g
a
b
e.g. another module
etc.
α/f,g
a
b
f
p
q
r
g
i
j
k
l
wrapped caller - but scope has changed
called functions implanted

Figure 14. Wrapping the caller as a set changes its scope

5. The lifecycle of an implanted asynchronous function is independent of the lifetime of any other machine. It is *only* destroyed when the transition to the terminator (symbol ⦿) takes place. Even if a parent is another function and is destroyed, the asynchronous function lives on until it transitions to its own terminator.

6. Additional intermediate notifications can be included as well as a final notification. Only the final notification corresponds to the function implantation being removed, but the way the system knows that the implantation is to be removed is by the fact that *the transition is to a terminator*. From an implementation perspective, even a final notification will behave like any other broadcast event.

7. As already mentioned, vents such as "pending" and "notify" need to identify the function they apply to by their name. But what if same function called twice in the same scope (there are various ways this might happen) - how to disambiguate broadcast events from each. Solution: Parameterize events to disambiguate.

8. More deeply nested parts of the implanted machine, if present, must use e.g. a `$$` scoping operator when targeting fired events at the caller.

9. Note how a generic transition on `in(f_notif) && in(g_notif)` and other specific transition paths out of the calling state are possible (though the above is a lambda event, not currently envisaged to be supported).

2.3 Mixed synchronous / asynchronous calls

If a transition contains calls to synchronous and asynchronous functions, implantations of the relevant kind can be provided systematically, as indicated in the following example. Events `σ,τ,υ` are synchronous functions, events `α β` are asynchronous functions, and other events (`ε ζ1-ζ6` and `return` events) are ordinary global events.

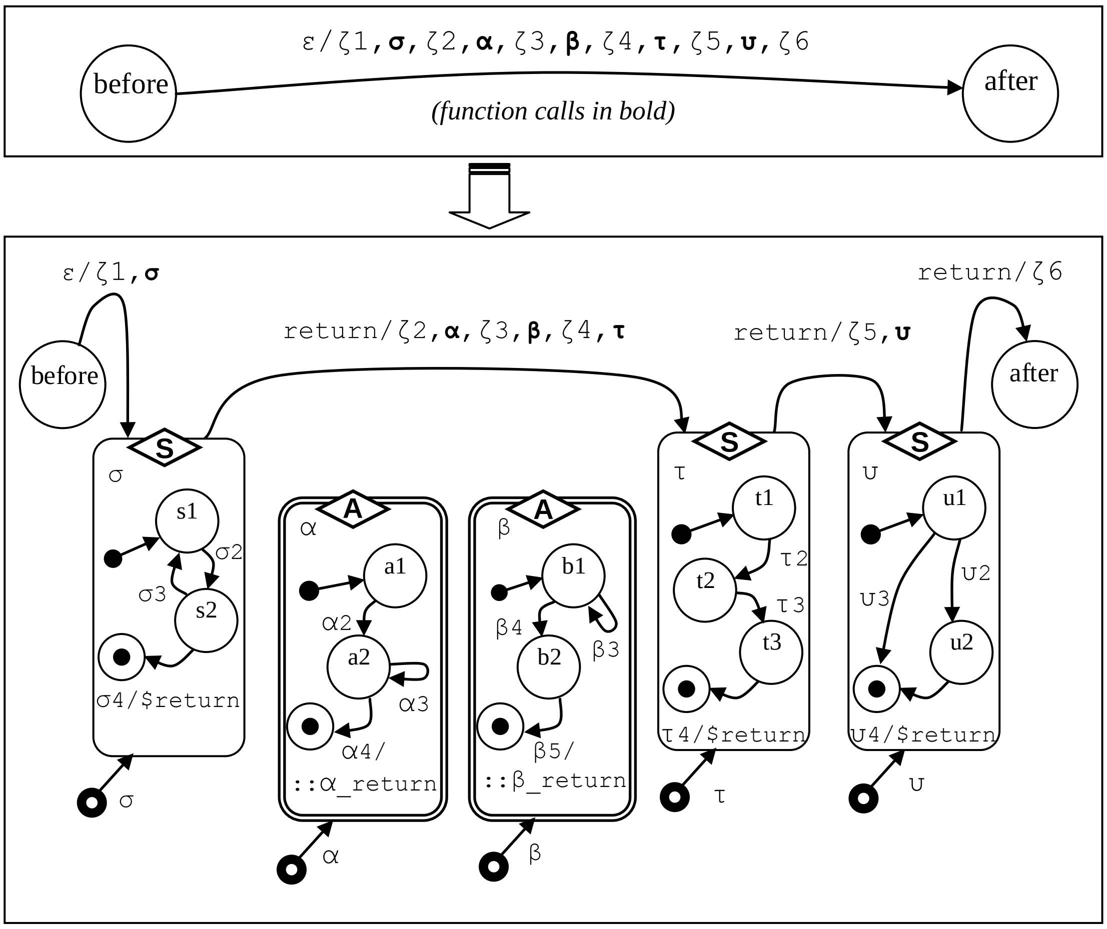


Figure 15. Mixed synchronous / asynchronous calls

Again, the synchronous functions could be implemented using the asynchronous technique.

2.4 Syntax issues for synchronous and asynchronous function calls

In order to have a focussed overview of all syntax issues resulting from all the techniques described in this report, syntax issues are discussed separately in chapter 5.

3. Parameter passing

Review of conventional parameter passing techniques

For imperative languages (such as Fortran, Algol, C++ and Java), functions, (or procedures), are defined with a *formal* parameter (or argument) list, e.g.

```
      int max(int p1, int p2, int p3) ...
```

When they are called, *actual* parameters are supplied, e.g.

```
      high=max(i,*pj,k+6);
```

The function might handle the parameters in various ways [vVliet]:

- Call by **value**. The parameters are evaluated and the results are assigned to local variables, which are accessed using the formal parameters. The caller's variables are not altered this way, except where a pointer is passed and an indirect assignment is made.
- Call by **value-result** (or call by copy-restore). As with *call by value*, the parameters are evaluated and the results are assigned to local variables, which are accessed using the formal parameters. At the end of the procedure, the actual parameters are assigned the final value of the local variables, so changing the value of the caller's variables.
- Call by **reference**. Here, the actual parameters are *evaluated once* and this result is substituted where the formal parameters occur. Any assignment to a formal parameter results in an assignment to the actual parameter, and so changes the caller's variables.
- Call by **name**. Here, the actual parameters are substituted *in unevaluated form* wherever the corresponding formal parameters occur. Any assignment to a formal parameter results in an assignment to the actual parameter, and so changes the caller's variables.

Fortran uses call by reference or call by value-result.

Java uses call by value, (though the caller's value is itself a reference).

C++ prefers the use of call by reference, illustrated below.

C++ program source:

```
#include <iostream.h>

void outint(int& i, int& j)
{
   cout << i << " " << j << endl;
   j++;
   cout << i << " " << j << endl;
}

void main()
{
      int a[5]={1, 11, 21, 31, 41}, j1=2;
      outint(a[j1], j1);
}
```

Program output:

```
21 2
21 3
```

What *call-by-name* output would have been:

```
21 2
31 3
```

Figure 16. Call by reference versus call by name

Problem statements

Fired events are used to model function calls, and it seems logical that the parameters of a function call should be modelled by parameters to the event. However, the caller and callee of a function call are typically in separate software components (client and server). The following questions arise:

- What is the recipient of a fired event? Is it the transition? Is it the target state? What if there is more than one target state? Is it some code executed as an action on the transition?
  ⇒In CHSM it is the implementation of the event handling routine. However, we would like the parameters to be accessible to the state model in its own modelling language, not in an embedded implementation language.
- What if several transitions are triggered by a parameterized event?
- What parameter passing method is applicable?
- What if the parameter is not immediately needed, but must be kept for future use?
- How can the parameters passed be retained in the server for its own use at any time, without the client having any knowledge of the server structure?

Solution: Parameter passing by *call-by-destination*

We model the parameterized events by a calling mechanism which we might denote by *call-by-destination* or *call-by-scoped-variablename-expression*.

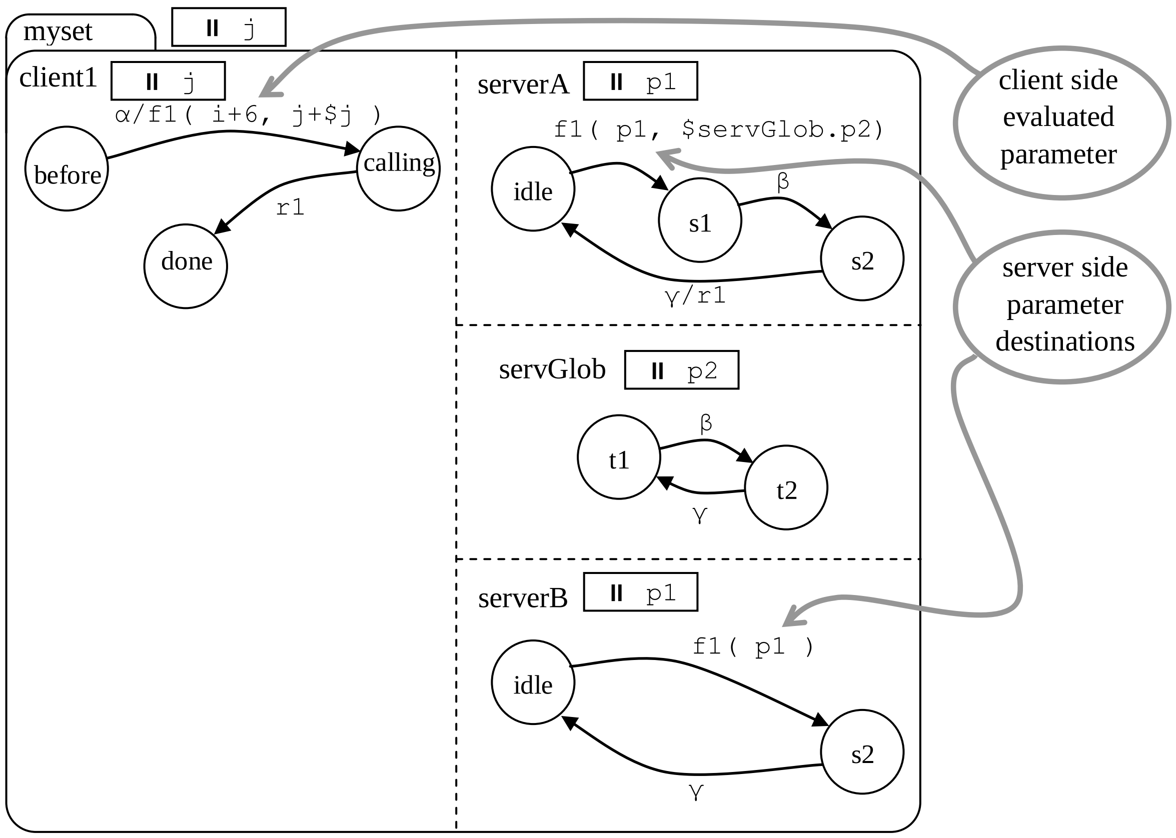


Figure 17. Parameterized event mechanism

Remarks

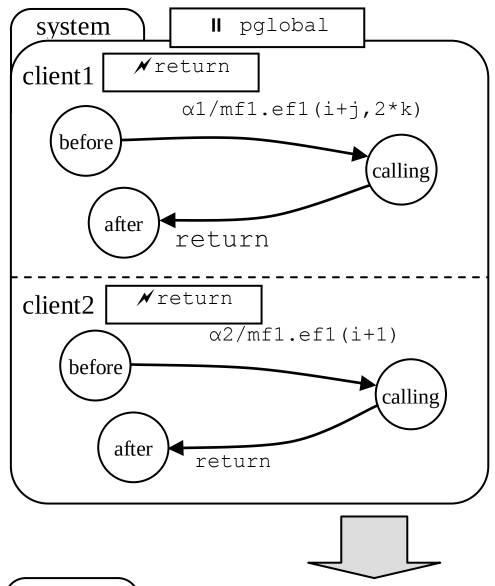
1. The basis is that an event can be generated with parameters [e.g. `fire f1(i+6,$j+j)`], and that the receiving transition directs the parameter values to destinations of its choosing, using scoped expressions which yield a machine path and name [e.g. `p1,$servGlob.p2`]. This recipient directs its first parameter to the local variable `p1` and its second one to the neighbouring set member's local variable `p2`.

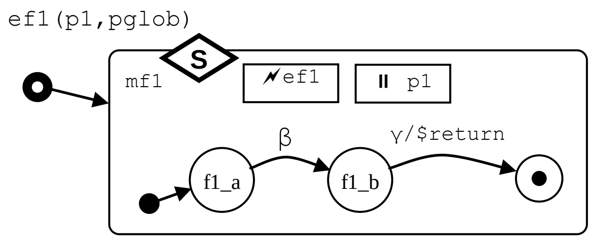
2. Not all parameters need be consumed. The serverB machine only uses the first parameter, and directs it to its own local variable p1 (which is not the same as serverA's p1).

3. There can be more than one machine that responds to an event. If the parameter destinations clash, then the execution order of responding transitions will determine the value of a shared variable at any time.

4. The parameter passing scheme described above does not correspond to any of the conventional schemes. One reason is that the receiving transition uses *actual* name-*expressions* rather than *formal* names.

Parameterized events employing implanted functions:





On function call, the model is temporarily dynamically modified into the following:

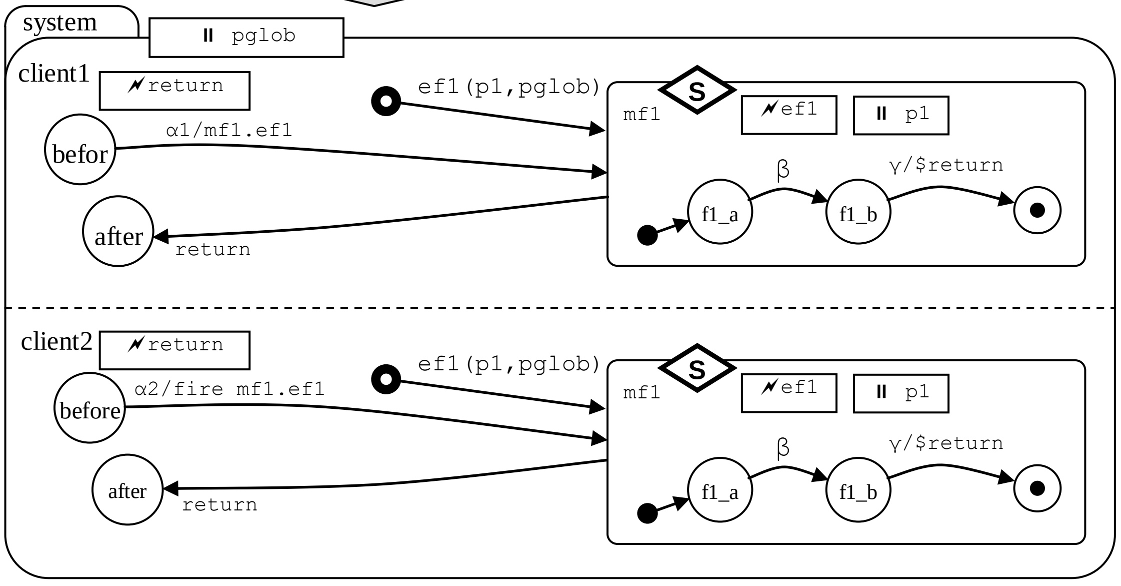


Figure 18. Parameterized events with implantable (dynamically modified) function models

Remarks

1. The two clients obtain an equivalent implanted model of the server function model. The function models direct one parameter to a local variable and one to a global variable. The function model does not show actual use of the variables `p1` (in two contexts) and `pglobal` - but use could be in the form of conditions on transitions or actions, further assignments, or further fired events.

2. Typically, when the functions return, any local variables etc. going out of scope will be removed. But any parameter directed to some variable that does not go out of scope will remain extant.

3. Servers with function models will typically be modelled by function templates *and* extra parts of a statechart. These might include a global set member and global data declarations.

4. The return or notification events of functions can supply a return parameter (or event several!) using the same mechanism as for parameters supplied on calling a function.

5. Syntax issues relating to parameterized events are covered in chapter 5.

4. Modelling a pump engine

4.1 Introduction

The concept of a pump is used in Windows, from which is has been adapted for use in MG-R [MGR-Pumps]. First, the Windows background is described, then the MG-R equivalent, and finally the problem is approached from a more general consumer-producer perspective.

Windows background, based on MFC/C++ documentation

In Windows, a message is sent to a Window using `PostMessage`, (or `SendMessage`, which will postpone return of control to the sender until the message has been completely processed by the recipient). The message goes to the thread that created the window. There will typically be several windows created by one thread. A message is sent to a *thread* (which may not own any windows) using `PostThreadMessage`. There is no `SendThreadMessage`. The programmer may simply define handlers for messages, but the message queue can be accessed directly. MFC's message loop lives in `CWinThread::Run`, which can be overridden. `Run` "provides a default message loop for user-interface threads. `Run` acquires and dispatches Windows messages until the application receives a `WM_QUIT` message". `Run` pumps messages while available by calling `PumpMessage`, which is an 'undocumented function', and so used or replaced by users at their own peril. It calls `TranslateMessage` and `DispatchMessage`, and so effectively converts messages into function calls.

MG-R pumps and pump engines

[MGR-Pumps] provides the following definitions:

> In a nutshell, a *pump engine* is the combination of:
> - a *message queue* (with a predefined message format);
> - a Real Time Kernel *task* (some people call this a *thread*);
> - a *message dispatch loop* running on this task.
>
> So each pump engine corresponds with an RTK task created specifically for that pump engine. This task cannot be used for other purposes; its sole purpose is to run the message dispatch loop. We shall postpone the discussion of the message format to a later section.
>
> On a pump engine, one must create one or more *pumps*. Each pump can be seen as a *logical* message queue (with a (simpler) predefined message format) to which a single function (the '*pump function*') is attached. Whenever there is a message in the queue, this function is called. The contents of the message are then provided to the function as a set of parameters.

MG-R pump engines can be seen as a way to allocate logical queues to physical threads by allocating the logical queues to pump engines. A pump engine can support several pumps (queues). The pumps have associated with them a single handler function – only one function can be called per pump. A message is sent to a pump by calling `PmpSend`; this function does not allow for failure to queue so queue length and average throughput must be correctly budgeted.

This can be expressed in UML as follows

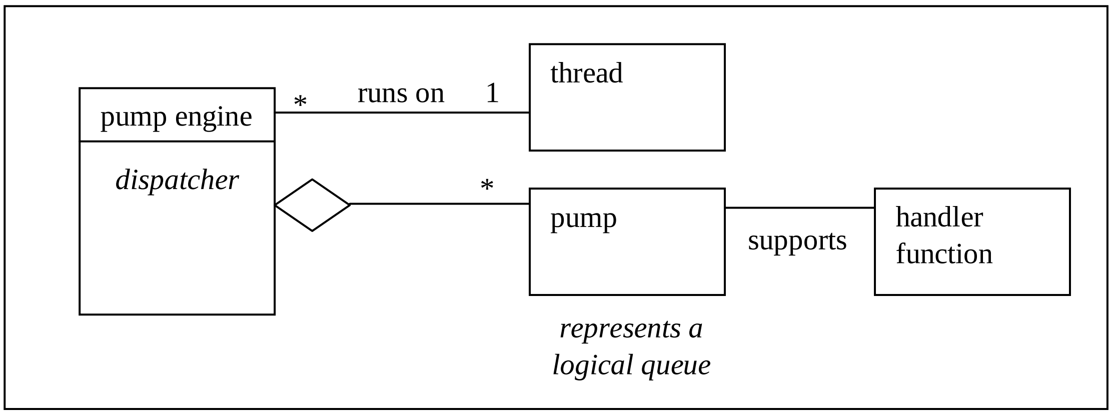


Figure 19. Pumps and Pump Engines - UML

or more pictorially

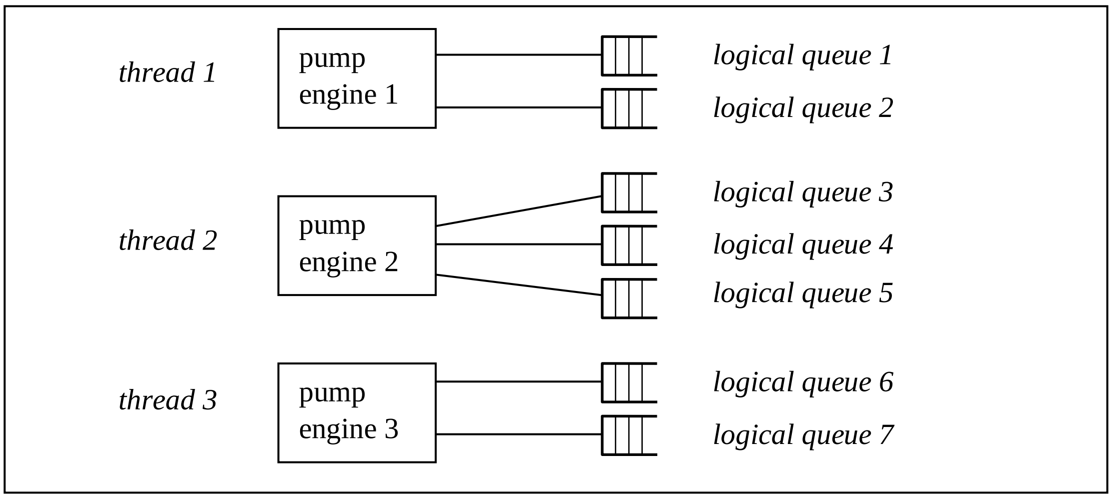


Figure 20. Pumps and Pump Engines - Pictorial

The Producer-Consumer Approach

Pump engines are a special case of the more general **producer-consumer** problem, and it from this perspective that we model them. A producer-consumer system is one where messages are produced by one thread or task and consumed by another. The threads will need to load-balance and synchronize.

A pump engine is a special case where the messages are function calls which just need to be executed, but in general the messages might not be so directly processable. We will model the messages as events.

Consumer-producer synchronization (and, further below, a semaphore implementation) are standard techniques, described for example in [SAD-RTS].

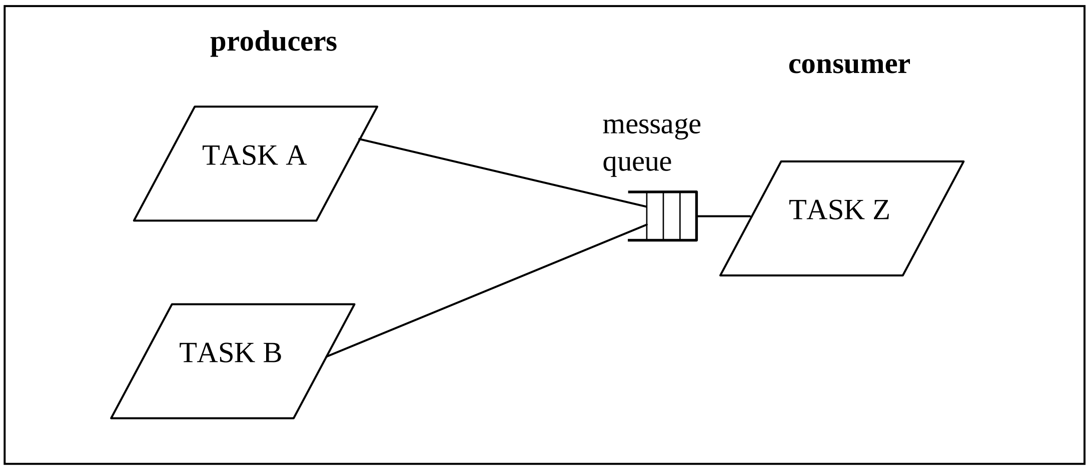


Figure 21. Producer Consumer synchronization

The operations of depositing messages in the queue and extracting them for processing will be protected by semaphores (described later).

There are two environments in which a producer-consumer system may operate:

- ***Unsuspendable Producer***. In real-time situations, such as when processing incoming TV frames at the frame-rate, it is not possible to slow the producer down. Other examples are incoming web content and Ethernet: the protocols do not allow for suspension of the producing machine. The queue sizes and average consumer processing rate must be adequate to maintain throughput. MG-R pump engines fall in this category.
- ***Suspendable Producer***. A non-time-critical producer can be suspended. An elementary example is a typical program producing output destined for a disk. The output will be buffered, and when the buffer is full, the *write* call will be delayed in returning control, waiting for the free buffer space. This delay stems the output production, so the system as

a whole is *levelled*, i.e. the producer is soon balanced in speed to the speed of the consumer.[2]

These two environments require separate state-based modelling, since the second one involves more states than the first. If a system is implemented with a suspendable producer, but nothing of interest ever happens when the producer is suspended, except to unsuspend the producer, then the system may be modelled as if it were the first case. Before approaching the second case, we will describe the internal semaphoring mechanism.

4.2 Unsuspendable producers

Messages are modelled as events. We require the following features in the state machine system:

- The availability of library functions to queue and dequeue event names. The functions might take two parameters: a queue name (= pump in MG-R) and an event name. Additional functions to test whether the queue is empty or not, and to remove duplicate entries, may be useful in certain circumstances, but are not needed for the basis.
- The ability to manipulate events:
  - to pass them as a parameter to a queue function call
  - to receive them in a dequeue function call
  - to fire an event received as a parameter.

In manipulating events, two approaches are possible:

1. **Implicit event referencing**. With this approach, we say that there is no need to have any special notation to take the name (or address) of an event, nor to dereference it. The following calls suffice:
   `queue(alpha), dequeue(evt), fire(evt)`
   Here, `evt` is implicitly a reference to an event. It will be identified as such and implicitly dereferenced, in order to fire the event. The function fire(...) is overloaded to take either events or references to events.

2. **Explicit event referencing**. In this case we are more precise about when a quantity is a an event and when it is some form of reference to an event. The reference could be made
   - by taking the *name* of an event, which might be represented by a *string* (with scope)
   - by taking the address of an event, e.g. `&alpha`, dereference example `*p_alpha`
   - by using the C++ reference operator (`&`) which is effectively internally an address-of operator, but a quantity `thing&` is type compatible with *`thing`* , i.e. with what it references, not type compatible with *`&thing`* (pointer-to-what-is-referenced).

For clarity we use the address-of and dereference convention in the diagrams, so we have

```
queue(&alpha), dequeue(p_evt), fire(*p_evt)
```

[2] A refinement to this model prevents thrashing. Thrashing in this case is the rapid alternation of allocating and releasing a small amount of buffer, thus incurring a high system overhead. Thrashing can be prevented by only returning to the producer when a fair amount of buffer is available.

However, we overload `fire`, so that we can write

```
      fire(p_evt)
```

Parameters to events

In either case the events and references to events must be parameterizable. Chapter 3 describes how events can be parameterized; an example using queued events is:

```
      queue(&eta(p1,p2+1)),
      dequeue(p_evt(q1,q2)),
      fire(p_evt(q1,q2))
```

Parameters could be stored in queued functions

- as expressions to be evaluated at function execution time
- as values evaluated at queuing time.

We opt for ***evaluation at queuing time***. Evaluation at execution time can be engineered by supplying pointers to data and evaluating the required expressions in the function body, rather than in the call.

Synchronous versus Asynchronous calls

The technique described below is applicable to synchronous and asynchronous calls, and we show a case of one of each kind of call being processed. Note that the terms synchronous and asynchronous are from the server's (i.e. consumer's) perspective (typically representing the consumer's thread), not with respect to the client's (i.e. producer's) perspective.

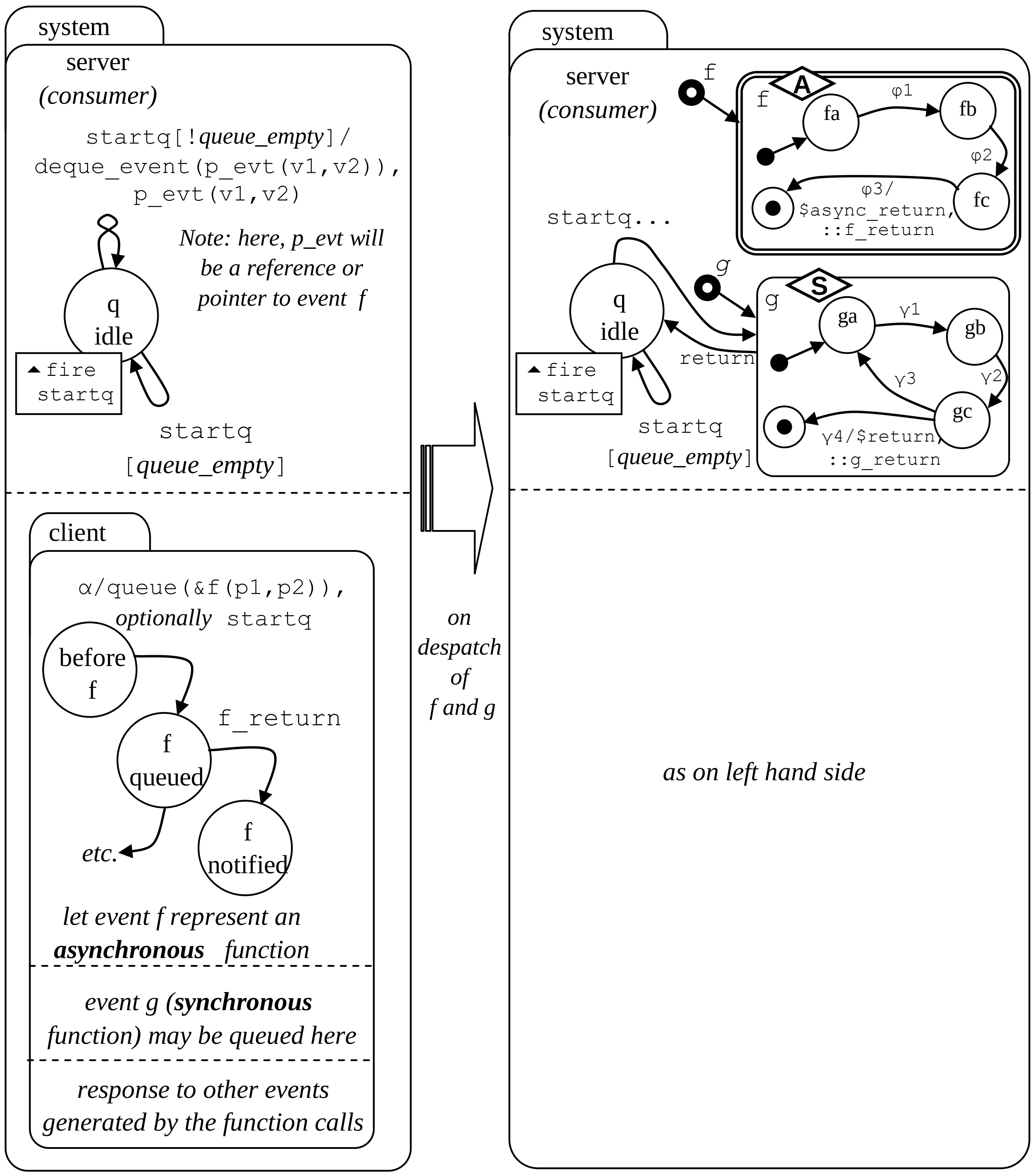


Figure 22. Unsuspendable Producer

Explanation and notes on unsuspendable producers

1. We employ the method of processing synchronous and asynchronous calls as described in the preceding chapters, but the initiator of the calls is now the consumer (state `q_idle`). There is a ***synchronous*** return event that is *local* to the consumer so that it can be returned to its idle state. A synchronous function completion will normally be visible as both a *local return* to the local caller and as a *global notification* to the client originator. This has been implemented by firing two separate events (`$return` and `::g_return`).

   The ***asynchronous*** notification event is *global*, (`::f_return`) so that it will be in scope for the client whever the client is in the scoping space. All functions need to use unique names for notification events. By unique names, we mean unique after any transformation due to component binding. Asynchronous functions may need to issue an additional return event, which should be called `$async_return`, to enable a distinction and not interfere with synchronous calls. This is used when considering nondeterministic suspension of a consumer later on.

2. Once started, this consumer engine will continue to dequeue and process events until the queue is empty. This is achieved by having an *orbital* self-transition on q_idle, with an `upon enter` action firing a new start event after each event has been despatched (whether representing a synchronous or asynchronous function). If the queue is empty, a *non*-orbital self-transition takes place and the queue is not re-started.

3. In this example we leave it to clients to explicitly start the queue. An automatic way to start the queue would be for the `queue` library routine to start the queue. There may be situations where fine control is needed of exactly when the queue is started, perhaps reflecting thread priorities, so it appears best to leave firing the start queue event to the user.

4. Note that a synchronous function call *blocks the consumer* until completed.

5. The technique can be generalized for several consumers by allowing for a queue name as an extra parameter to a `queue` call.

6. Note that the client, having queued an event, is not blocked in any way, whether the call will be run synchronously or asynchronously by the consumer.

4.3 Suspendable producers

Figure 23, based on [SAD-RTS] course material, illustrates how messages are deposited and extracted using semaphores. It shows the two reasons why a client might be suspended:

- Lack of buffer space in the queue. In this case, the client will have to wait until the consumer has consumed one (or more) messages.

- Consumer is busy with buffer manipulation. This is typically a very short operation as first-in first-out queues can be implemented such that for any length of queue a deposit or extract process takes a fixed amount of processing which is independent of the length of the queue. US patent 6,018,612 (Thomason & van Loon) describes such a method (but this is not part of the invention). In the scheme described there, buffer insertion requires following one fixed pointer one step, testing a value for zero, and making two assignments. Buffer extraction requires following one fixed pointer one step, testing for zero, and making one assignment. Inserting into an *empty* buffer, and extracting the *last* message, will show an affirmative test-for-zero condition and will require minor variations.

The lack-of-buffer situation can lead to deadlock, e.g. if a message being consumed synchronously requires an event to progress, but where that event must come from the blocked client.

The buffer manipulation operation in itself cannot lead to deadlock, as it is just a matter of time before the critical operation is complete and the semaphore is released. It is probably often adequate to consider the theoretical *buffer-manipulation-busy* state as indivisible, so that it would not need a separate modelled state.

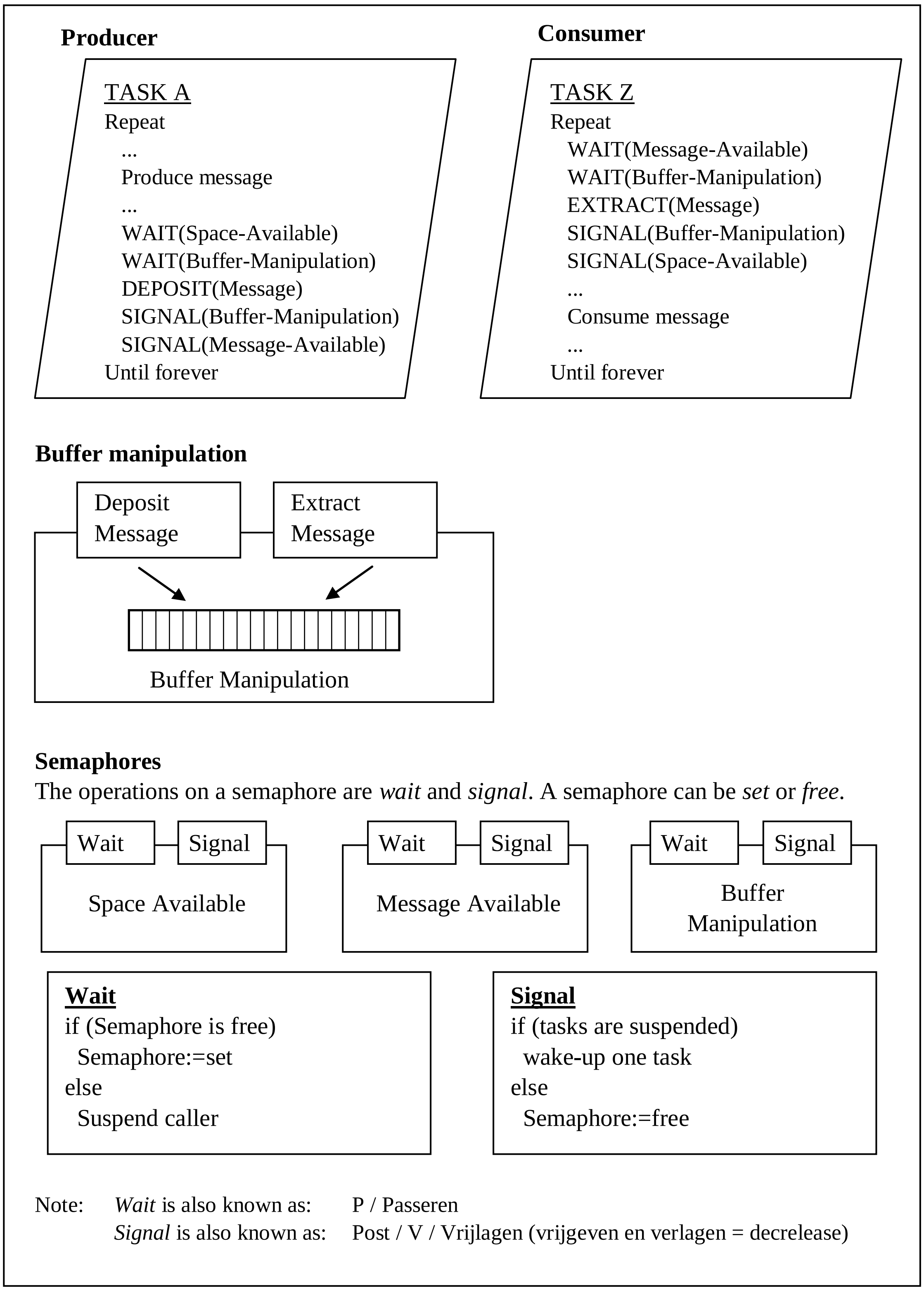
The fact that the *consumer* may be waiting for a message has been handled by the `startq` event. The consumer may also be waiting for the buffer manipulation semaphore to be released, but again this is just a matter of time, and it may well be acceptable to model it as an indivisible situation.

There are alternative abstractions[3] of, and implementations[4] of the consumer-producer synchronization problem, but a client wait situation will be a common result.

In our consideration of the situations we simply allow for modelling of one client waiting state.

[3] E.G. Petri-nets, Ada's Rendez-vous

[4] Semaphore-style synchronization can be implemented in, say, C without any operating support. References: [Harel] and http://courses.cs.vt.edu/~cs3204/spring2001/cstruble/notes/chapter8.pdf



Producer

```
TASK A
Repeat
  ...
  Produce message
  ...
  WAIT(Space-Available)
  WAIT(Buffer-Manipulation)
  DEPOSIT(Message)
  SIGNAL(Buffer-Manipulation)
  SIGNAL(Message-Available)
Until forever
```

Consumer

```
TASK Z
Repeat
  WAIT(Message-Available)
  WAIT(Buffer-Manipulation)
  EXTRACT(Message)
  SIGNAL(Buffer-Manipulation)
  SIGNAL(Space-Available)
  ...
  Consume message
  ...
Until forever
```

Buffer manipulation

Semaphores

The operations on a semaphore are *wait* and *signal*. A semaphore can be *set* or *free*.

```
Wait
if (Semaphore is free)
  Semaphore:=set
else
  Suspend caller
```

```
Signal
if (tasks are suspended)
  wake-up one task
else
  Semaphore:=free
```

Note:	*Wait* is also known as:	P / Passeren
	Signal is also known as:	Post / V / Vrijlagen (vrijgeven en verlagen = decrelease)

Figure 23. Detail of Synchronization using 3 semaphores

What needs to be modelled for a suspendable producer?

If a consumer *may* become suspended, and from a testing perspective the occurrence of this situation can be engineered in the SUT (System Under Test), then the state-based model will need to allow for precise control of whether suspension takes place or not. This can be done by using

- a boolean variable to determine whether suspension takes place
- an extra event, fired externally to simulate a repeated attempt at queuing the event to be queued.

We call this the *deterministic* case.

If the SUT cannot be controlled as to whether the consumer is suspended, then a *nondeterministic* model may need to be made. This is considered after the deterministic case.

An example of deterministic modelling follows, with `q_ok` as the boolean variable to control whether the consumer is suspended, and β as the extra event.

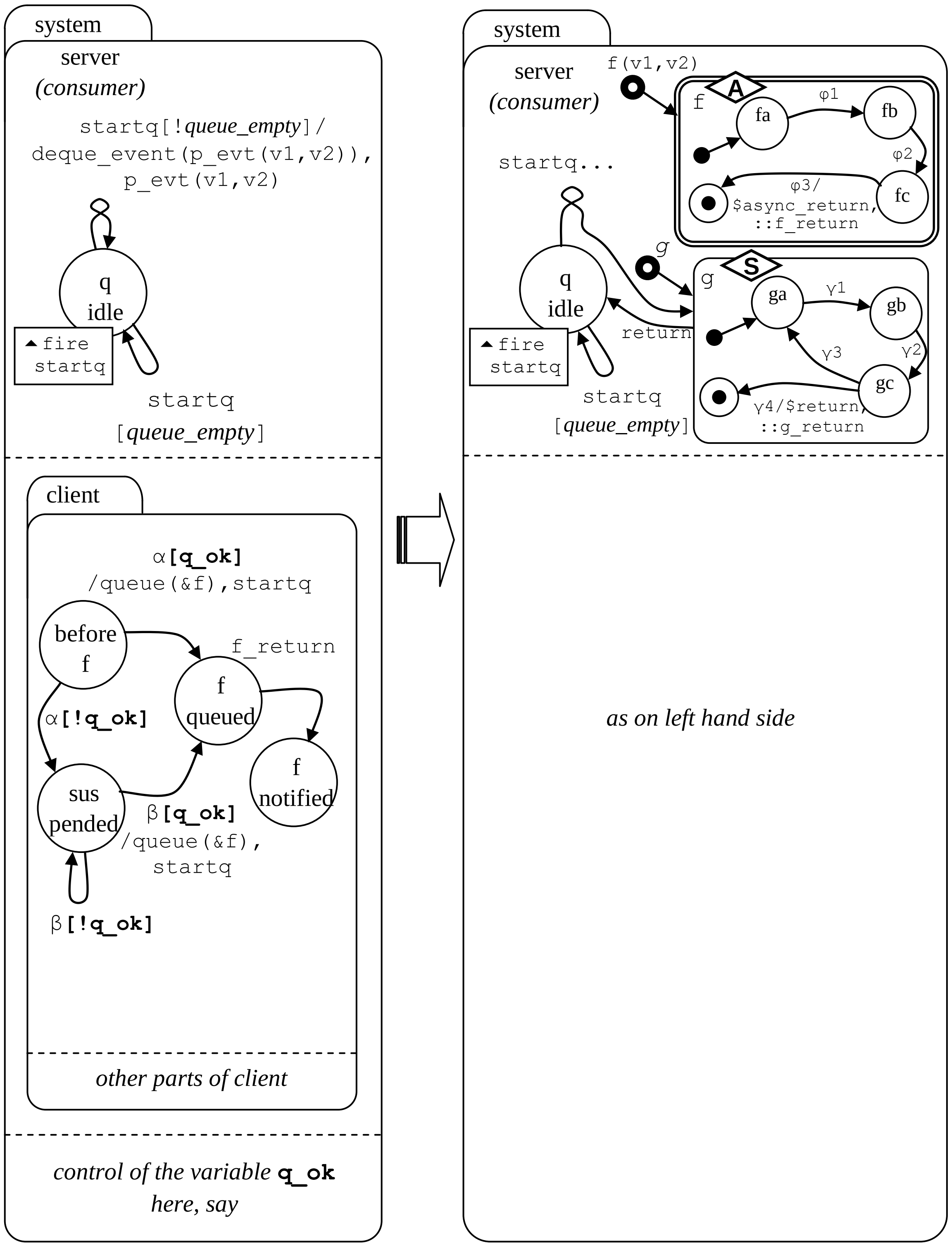


Figure 24. Deterministic suspension of consumer

***Nondeterministic* modelling of a suspendable consumer**

If a consumer *may* become suspended, but from a testing perspective it is not possible to predict whether this will happen, then a nondeterministic approach may be required.

An example is given in the following figure.

Explanation and notes on a nondeterministically suspendable consumer (see figure following)

1. In this figure, function `d` is some previously activated asynchronous function, and it is when this completes that waiting clients *may* be able to queue their messages.

2. Event α is the event that makes the client want to queue a message. Event `::server.async_return` is generated outside the client and provides another opportunity for the client to queue a message after having previously failed. Note the fork-nondeterminism on events α and `::server.async_return`.

3. Asynchronous functions need to fire a `$async_return` event (as used above) in addition to their globally unique notification. This is used to effect a retry-to-queue by clients that failed to queue something the first time. If there are several servers for different queues, then clients must make sure that their 'retry' events are the ones from the server that will handle the event being queued by the client.

4. This technique could lead to many deadlocked worlds where the client is (unrealistically) suspended, because there are no actively dispatched functions claiming the buffer space. However, there will always be *one* world that corresponds to the observed behaviour of a correctly-behaving client.

5. An alternative technique for retrying to queue is to *periodically retry*. This can be very inefficient, since many retries may be made when there is no chance of success (because nothing relevant has happened), or conversely a golden opportunity may arise and not be taken until after a considerable delay. This technique is not further considered in this report.

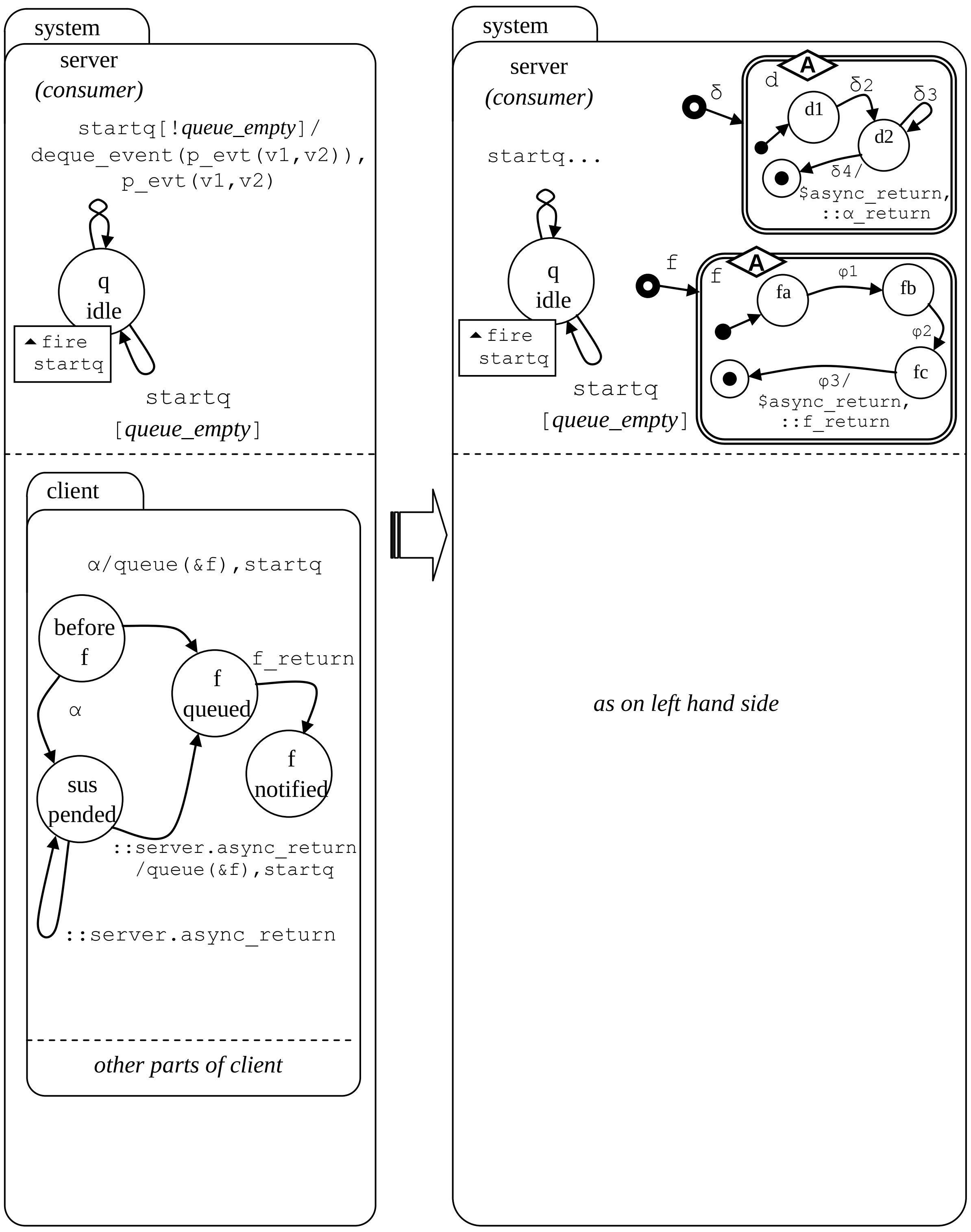


Figure 25. Nondeterministic suspension of a consumer

5. Summary of syntax issues

Bound synchronous function calls

Functions will initially be implemented in PROLOG according to the STATECRUNCHER function API. Future development could be

- to support C more fully
- or to interface to dynamic link libraries, (giving a measure of language independence)
- to interface to COM components (giving better language independence)

Synchronous versus asynchronous function calls

The distinction between synchronous and asynchronous functions is in made on the ***server-side***, i.e. in the representation of the function model, not the transition into it. However, the client will need to know whether it is calling an synchronous or asynchronous function because the client models responding to events are different, the asynchronous case requiring an explicit intermediate 'called' state.

Unbound synchronous function calls

- Implantable state models of the state behaviour are compiled as (freestanding) standard statecharts with the keywords **`synchronous`** `statechart`
- *See below for syntax relevant to synchronous and asynchronous function calls.*

Asynchronous function calls

- Implantable state models of the state behaviour are compiled as (freestanding) standard statecharts with the keywords **`asynchronous`** `statechart`
- There is no need to syntactically identify notifications as such as the pertinent factor is that they are on a transition to a terminator.
- *See below for syntax relevant to synchronous and asynchronous function calls.*

Synchronous and asynchronous function calls

- Transitions to function calls are recognized by the event name on the transition matching the (freestanding) statechart name
- Initially, only one transition on creation will be supported, so no new syntax is needed to specifically identify such transitions in the function model.
- The end state is recognized by the keyword **`void`**.

Parameterized events

- Fired events can be followed by a parameter list, e.g. `fire alpha`**`(v1+1,v2)`**
- Events that trigger transitions can be given a destination parameter list, e.g. `alpha`**`(p1,$$p2)`**

- Synchronous and asynchronous functions can be followed by a parameter list after the statechart name and the child-list , e.g.
  ```
      asynchronous statechart sc(child1,child2)(::q1,child1.q2)
  ```

Pump engine modelling

For pump engine modelling, the extensions needed to the state-modelling tool STATECRUNCHER are:

- Library functions to support FIFO (First-In First-Out) queues: an **`enqueue`**, a **`dequeue`** a **`test-queue`**, and a **`remove_duplicates`** routine. Others can be added as the need arises. Unless they require some new *type* of parameter, they do not affect the underlying STATECRUNCHER syntax.
- **Pointers to data**, so that function parameters can be evaluated at function execution time rather than function queuing time. This is a matter of extending the operator set and expression evaluator.
- **Pointers-to-events**, or references to events. Again, this is a matter of extending the operator set and expression evaluator. The function fire will be overloaded to accept, and automatically dereference, pointers to events.

None of these requirements materially impact the STATECRUNCHER language as such.

Models of synchronous and asynchronous functions will normally adhere to the convention that they provide a global return event (with a unique name) *and* a local return event (`$return` and `$async_return`) respectively.

Models allowing for nondeterministic suspension of a consumer may need to make provision for retrying-to-queue. They can do this by providing a transition that triggers off `async_return` events in the scope of the pump engine that will handle this consumer's message.

6. References

STATECRUNCHER documentation and papers by the present author

Main Thesis	[StCrMain]	The Design and Construction of a State Machine System that Handles Nondeterminism

Appendices

Appendix 1	[StCrContext]	Software Testing in Context
Appendix 2	[StCrSemComp]	A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 3	[StCrOutput]	A Quick Reference of STATECRUNCHER's Output Format
Appendix 4	[StCrDistArb]	Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 5	[StCrNim]	The Game of Nim in Z and STATECRUNCHER
Appendix 6	[StCrBiblRef]	Bibliography and References

Related reports

Related report 1	[StCrPrimer]	STATECRUNCHER-to-Primer Protocol
Related report 2	[StCrManual]	STATECRUNCHER User Manual
Related report 3	[StCrGP4]	GP4 - The Generic Prolog Parsing and Prototyping Package *(underlies the STATECRUNCHER compiler)*
Related report 4	[StCrParsing]	STATECRUNCHER Parsing
Related report 5	[StCrTest]	STATECRUNCHER Test Models
Related report 6	[StCrFunMod]	State-based Modelling of Functions and Pump Engines

References

[vAntwerpen]	Hans van Antwerpen DVP2 Interfaces: Rules and Guidelines Philips HVE Software Architecture Board Presentation, Jan. 2001
[CompBinding]	G.G. Thomason Component binding in Composite Models for State-Based Testing. Philips PRL Technical Note 4102, August, 2001
[Harel]	D. Harel Algorithmics, The Spirit of Computing Addison Wesley, 1987. ISBN 0-201-19240-3
[Koala]	R. van Ommering, F. van der Linden, J. Kramer, J. Magee The Koala Component model for Consumer Electronics Software IEEE Computer, March 2000, pp. 78-85.
[KoalaYP]	http://pww.natlab.research.philips.com:25147/koala/
[Kunst]	P.J. Kunst Interface-based programming in C using vtables Philips Nat. Lab. Technical Note 2001/245, July 2001
[MGR]	R. van Ommering The MGR Software Architecture http://nlww.natlab.research.philips.com:8080/ research/ist/people/ommering/docs/1999/99PSC_MGR.pdf
[MGR-Pumps]	Rob van Ommering On Pumps and Pump Engines http://pww.natlab.research.philips.com:25147/mgr/arch/pumps/index.htm *(password required)*
[SAD-RTS]	Structured Analysis and Design for Real-Time Systems (Course material) Philips Centre for Technical Training

[Trew] Tim Trew
Software Component Composition - Still "Plug and Pray?"
Proceedings of the 6th Philips Software Conference, February, 2001

[vVliet] Hans van Vliet
Software Engineering Principles and Practice, John Wiley;
ISBN 0 471 93611 1