A Semantic Comparison of STATECRUNCHER and Process Algebras

Graham G. Thomason

Appendix 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

This paper discusses the essential differences in the STATECRUNCHER approach to composition and synchronisation of processes, and to nondeterminism, to that of the process algebras CCS and CSP. It is a pre-requisite to the papers mentioned below, covering ground common to them.

In *separate papers* a more detailed discussion of specific case studies, taken from the CCS and CSP literature, is given. Those papers show working STATECRUNCHER models of the systems, covering their statechart diagram, source code, and output from sessions running the models. A comparison of the STATECRUNCHER model with the CCS or CSP specification is given. An additional study shows how a Z specification relates to STATECRUNCHER concepts. The case studies in those papers are:

- The Distributed Arbiter System in CCS [StCrDistArb]
- The Dining Philosophers in CSP [StCrMain]
- The Game of Nim, specified in Z [StCrNim]

Reminder of the motivation for STATECRUNCHER

STATECRUNCHER was built for the purposes of providing an oracle to state-based tests. It forms part of a tool chain for *testing an implementation* of a system, i.e. for determining whether the implementation under test behaves according to its specified state behaviour, even when it is nondeterministic. STATECRUNCHER *does not generate tests*; it co-operates with a test generator in a tool chain.

Table of Contents

1. Comparison of terminology

In STATECRUNCHER terminology the main concepts in a *statechart* are

- states
- events
- transitions
- actions
- traces
- variables
- assignments to variables

In order to concentrate on the essentials, we do not discuss here other refinements such as multiple target states, orbital transitions, conditional transitions, conditional actions, references to state occupancies, meta-events, parameterised events, and upon-entry/upon exit actions. These are described in [StCrMain, StCrParsing]. Mention will be made, however, of PCOs (points of control and observation), as a fixed attribute to an event.

The STATECRUNCHER terminology is different to that of CCS and CSP. STATECRUNCHER *actions* and *traces* are not the same as those of CCS and CSP. The STATECRUNCHER terminology corresponds more closely to that of tools used within Philips over the years such as [CHSM] and [TorX]. For this reason, a comparison is now offered. We start with a review of STATECRUNCHER terminology.

A very simple STATECRUNCHER model is shown in the figure below:

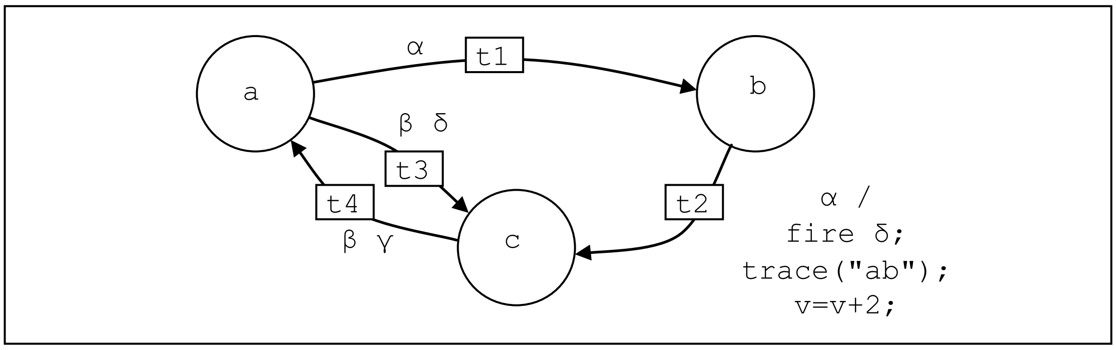

Figure 1. States, events, transitions and actions

The above diagram models a system as having:

- three states: `a`, `b` and `c`
- four events: α, β, γ, and δ
- four transitions: `t1`, `t2`, `t3` and `t4`
- three actions: `fire δ` and `trace("ab")` and `v=v+2`

At any one time, a system modelled by the above state-transition diagram will be in one and only one state. That state is called the *occupied* (or active) state. The others are *vacant* (or inactive). Since in general, in more complex models, several states can be occupied (due to parallelism and hierarchy), we speak of an *occupancy configuration*.

The main relationships between these are expressed as follows:

- an event *triggers* a transition, for example, α triggers `t1`.
- a transition *occasions* any actions on that transition. There are actions on transition `t2`.
- an action does one of the following:
  - *fires* an event, for example an action on transition `t2` fires event δ. In the above model, nothing responds to δ, but if there were a parallel part of the statechart, or even another transition from state `b` triggered by event δ, the response would be made.
  - *generates* a trace
  - *makes an assignment* to a variable

When a transition occasions an action, we may speak of the transition itself firing the event, generating the trace, or making the assignment, e.g. "transition `t2` fires event δ".

In STATECRUNCHER, an *event* may occur at any time, but a *transition* will only take place if the source state of the transition is occupied. STATECRUNCHER has commands to tell it to provide the set of all events and the set of *transitionable* events.

STATECRUNCHER *traces* are specific outputs on a transition that the modeller decides to record, so that the model can output them on request. They typically correspond to observable outputs of a system under test, and are important in black-box testing, where the states and internally generated events cannot be observed. On this basis, in the above figure, only transition `t2` produces output; the others are silent, and the only way to try to prove they have taken place is to drive the machine on through `t2` by an event sequence.

Summary of approximate equivalences

STATECRUNCHER	CCS	CSP
state	(state of an) agent. [Milner, p.19]: "Rather than distinguishing between two concepts - agent and state - we find it convenient to identify them, so that both agent and state will always be understood to mean agent in some state."	process
event	action, handshake [Milner, p.17]	event
transition	transition, as in A' ^{geth} 2 A [Milner, p.38]	transition [Hoare, p.34], as a pictorial aid. Note: *x 1 P* describes an agent that can engage in event *x* and become agent *P*.
action	*probably best modelled as an output action*	*probably best modelled as an output action*
trace	*probably best modelled as an output action with which a user can engage*	*probably best modelled as an output action with which a user can engage*
(sequence of processed events)	trace	trace

Table 1. Approximate equivalences in terminology

This table serves as a rough guide and an alert that the terminology is used differently in the different systems. The differences in approach will become more apparent as processes, and their composition, are discussed.

The ways in which *nondeterminism* is handled by the different systems is considered in section 3.

2. Composition of processes

In CSP and CCS, processes are combined by sharing events.

For ***CSP***, Hoare says [Hoare p.65-66]

> When two processes are brought together, the usual intention is that they will interact with each other. These interactions may be regarded as events that ***require simultaneous participation of both the processes involved***.

The CSP operator for *composition* is ||. The expression *P*||*Q*, is initially introduced for the case where processes *P* and *Q* have the same alphabet [Hoare, p.66 l.8], i.e. the same set of events, though this is relaxed in a generalisation [Hoare, p.69]. We will adopt the generalised version of the operator in our discussions that follow as in so many realistic examples interacting processes only share *some* of their events, namely the ones where they engage each other. (Hoare perhaps unwittingly uses the generalised operator before introducing it, in his example X2 [Hoare, p.66], where the alphabet of *FOOLCUST* lacks event *small*, which is in the alphabet of *VMC* [Hoare, p.30]). More than two processes can be assembled using this commutative and associative operator, e.g. *P*||*Q*||*R*.

CSP also has an *interleaving* operator |||, [Hoare, p.119]. In the expression *P*|||*Q*, only one process will engage in any action. If both processes can engage in an action, a nondeterministic choice is made between them. There is no notion of processes engaging one another.

In ***CCS***, the composition symbol is | , as in *Jobber* | *Hammer*, [Milner, p.29], where these particular agents share events for picking a hammer up and putting a hammer down. It is possible to have several instances of one agent, giving an expression such as *Jobber* | *Jobber* | *Hammer*. CCS allows two (and only two) processes to synchronise by performing an action and a complementary action together (e.g. c and c3), regarded as the handshake action τ. Milner describes the handshake and composition operator "|" along the following lines [Milner, p.39]:

if A' ^{c3} 2 A   and B ^{c} 2 B'
then
A' | B $^{\tau}$ 2 A | B'

The event τ is internal to the composite agent [Milner, p.39], and it is used to describe the internal synchronisation action of any pair of complementary actions.

We note that in CCS, event τ does not necessarily take place when it potentially can. The composite agent *may* perform a τ action which results from (c,c3) communication between its components [Milner, p.40].

Restriction on c, (and so implicitly on c3), which is denoted by \{c} or just \c, excludes independent execution of c and c3. It is a nondeterministic eventuality as to whether event τ is actually performed.

STATECRUNCHER will allow parallel parts of a statechart to share events, but this is not the same as CCS synchronisation, because there is no notion of event complements. STATECRUNCHER composition can best be achieved with a *fired-call-event / fired-return-event paradigm*, as follows. We then consider this composition paradigm in relation to process algebras.

STATECRUNCHER's composition paradigm

The standard paradigm for composing software components using STATECRUNCHER is to regard one component as a client (or caller) and one as a server (or callee). An event is fired by the client to call the server, and a return event is fired by the server to the client.

This has been elaborated on in detail, with some novel ideas, in [StCrFunMod].

The following figure illustrates the principle:

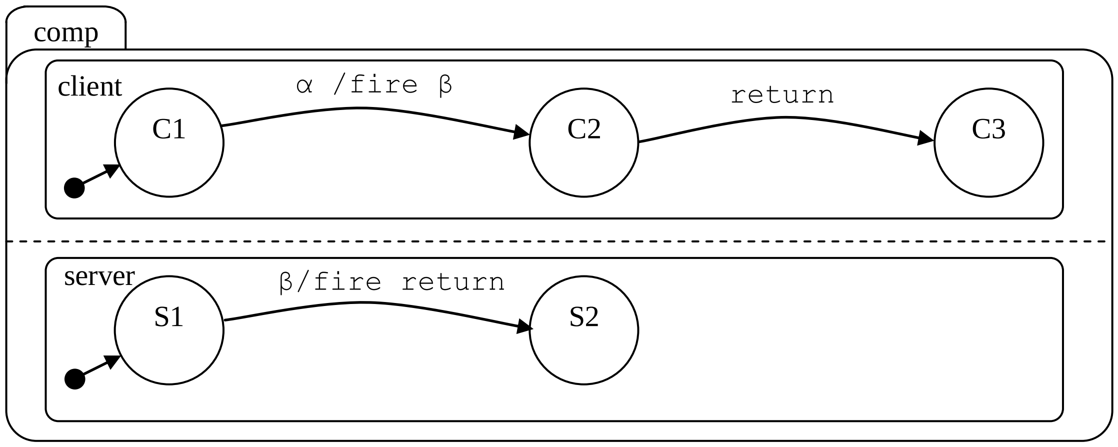

Figure 2. Client-server composition in STATECRUNCHER

STATECRUNCHER's composition paradigm is closely analogous to the function call and return of imperative languages such as ‘C’. The *making* of the function call is modeled by a fired event, the *response* to this is modeled by a transition on the event that was fired. The *return statement* is modeled by fired return event, and the *response* to this is modeled by a transition on the return event. If there are many such calling sequences in a model, return names can be made unique to a server function by affixing the function name to the event (e.g. `return_max`) or by putting the return event in a sufficiently local scope (using

STATECRUNCHER's scoping capabilities - described in [StCrMain] - but not further discussed here).

From the initial configuration, when event α occurs, the client transitions to state *C2* and fires event β. This causes the server to make a transition. In this example the server has immediately completed its work, and it immediately fires event *return*. This causes the client to transition to state *C3*. The whole sequence is regarded as atomic to STATECRUNCHER, in the sense that no other event can interrupt it.

In STATECRUNCHER, the interaction on event α definitely takes place. There is no nondeterminism involved as in the case of a τ event in CCS, where the transition only *may* take place. This is because we are typically modelling function calls and their return. However, if in CCS the only event that can take place is τ, then it can be argued that it should be considered deterministic.

We would not expect event β to be generated except by a client of the particular server. The name β would typically correspond to a server function name. There might, however, be several clients. We consider that situation later.

If the composition is a server to some higher level component, then the `α/fire β` construction will be repeated at a higher level (e.g. `δ/fire α`). It need not concern us as it is a repeat under different names of what we have seen. Alternatively, α is at the top level and is user supplied.

The transition semantics are important to allow this paradigm to work. A transition is taken to completion before its actions are executed. This ensures that no participating transition is blocked by its source state not being ready (i.e. occupied) for execution. So the transition on event `return` can take place because its source state C2 will be occupied.

The individual models of the client and server can be experimented with separately under STATECRUNCHER. But in the absence of, say, the server, an event *return* for the client will need to be given at prompt level by the user. Events should be attached to a point of control and observation (PCO). Event β and *return* would be put on an inter-component PCO, which can only be used in *module* testing. Under *integration* testing, this PCO and the events on it become internalized, or restricted or hidden, in CCS/CSP terminology, as the composition only admits to events such as event α.

The STATECRUNCHER composition paradigm is analogous to the composition of Process X-machines, as described in [Stannett]. The paper has:

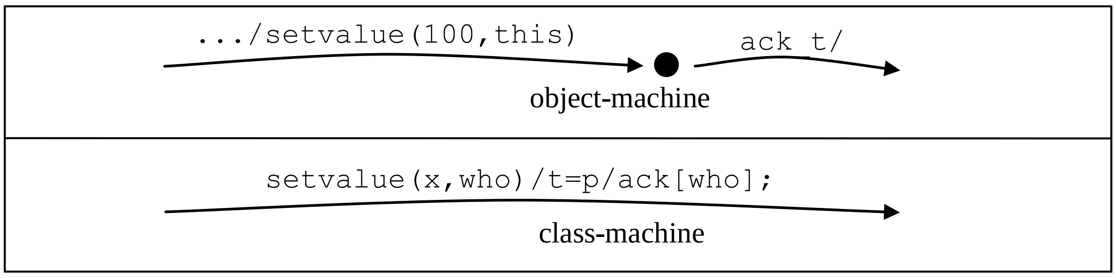

Figure 3. PXM assignmment to a static class variable by an object

The STATECRUNCHER analogue is:

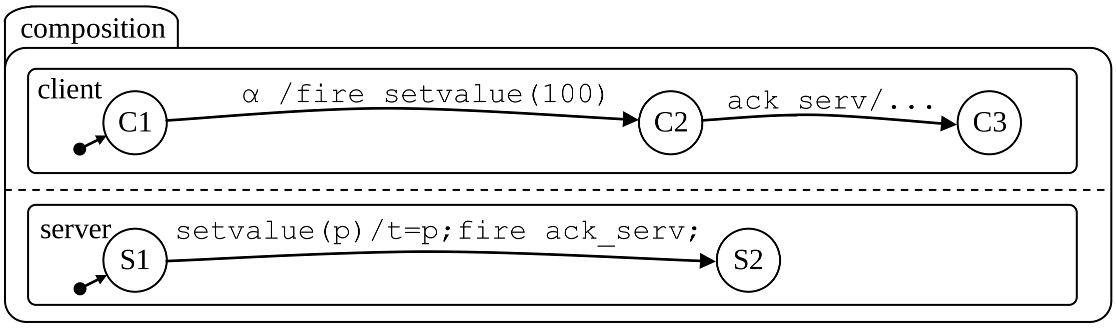

Figure 4. STATECRUNCHERs composition paradigm making an assignment

Here, we have not made the `ack_serv` event unique to the specific caller as in the paper (the *`this`* keyword). Since this server does not support recursion, the server can only be serving one client at a time, so it is sufficient for `ack_serv` to be unique to the *server*; it cannot then be confused with the acknowledgement from any other server serving a different function. In [StCrFunMod], we propose a composition mechanism for recursive state machines, where the returned acknowledgement need not have a unique name at all, and targets its caller by means of scoping operators.

The STATECRUNCHER composition paradigm has been used to compose models of Koala components [Koala]. Koala is a static component binding tool used by Philips for embedded software. STATECRUNCHER is being used to test some Koala television components. In Koala representation, the component binding would be drawn thus:

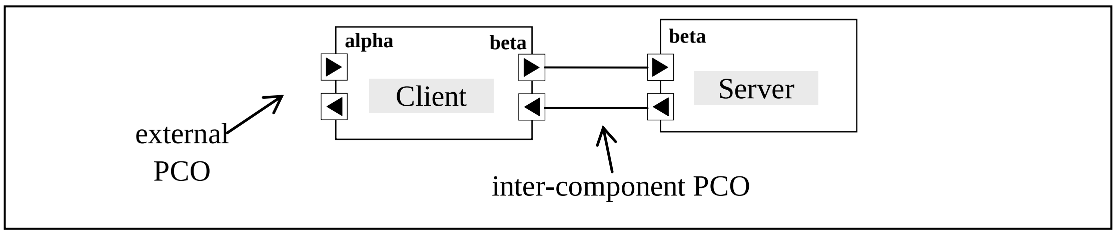

Figure 5. Koala components

A more realistic server in practice

Figure 2 is conceptually the simplest possible example of client server composition. There could have been additional states and transitions in the server before the return event was given, in which case the client would be in state *C2* for a while until the server was able to fire *return*. In such a case, the server might look like this:

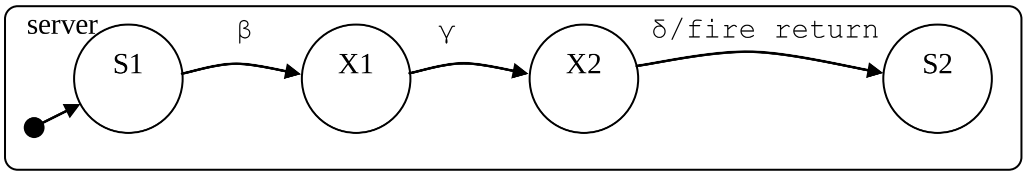

Figure 6. Server with intermediate states

It would be normal for a server to end up in its default state when a client has been served and returned to, as follows:

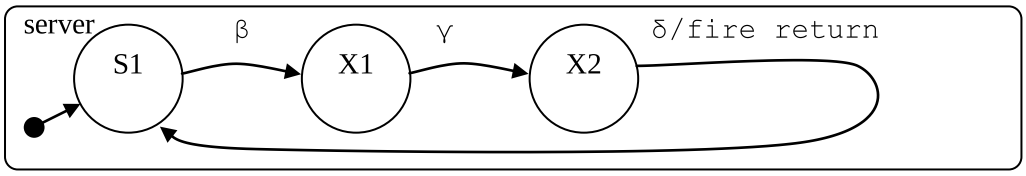

Figure 7. Server ending up in default state

Referring back to Figure 2: we do not return to the default state (S1); instead we are in a different state (S2) after the call, as this makes the calling paradigm as such a little clearer. One could think of the server as requiring some form of reset before it can be used again (not shown in the model).

Parameters can be passed back and forth by means of STATECRUNCHER's parameterized event mechanism. The issues of multiple clients, unique naming, and re-entrant or recursive calls is dealt with in [StCrFunMod].

Under this general system, a model of the server can be combined with *any* client that calls it with the agreed event β and which expects a return event *return*. Similarly the client could be combined with a different server as long as the interface was defined in the same way.

STATECRUNCHER's composition paradigm and process algebras

Let us examine the properties of Figure 2 and consider how to model it in a process algebra.

It has three STATECRUNCHER events, α β and `return`. It has two STATECRUNCHER actions, `fire β` and `fire return.` Questions we will be considering are:

- Should the *fire* actions be considered events in a process algebra?

- If so, should some of the events be paired off, into (event, complementary-event) pairs?
- Which events should be restricted (recalling that β and `return` are on an inter-component PCO)?

We first consider the composition from a CCS perspective.

The composition could be modeled using the CCS (combinator [Milner, p.68], giving Client(Server. We show this using the CCS port diagrams [Milner, p.17], which are similar to Koala diagrams.

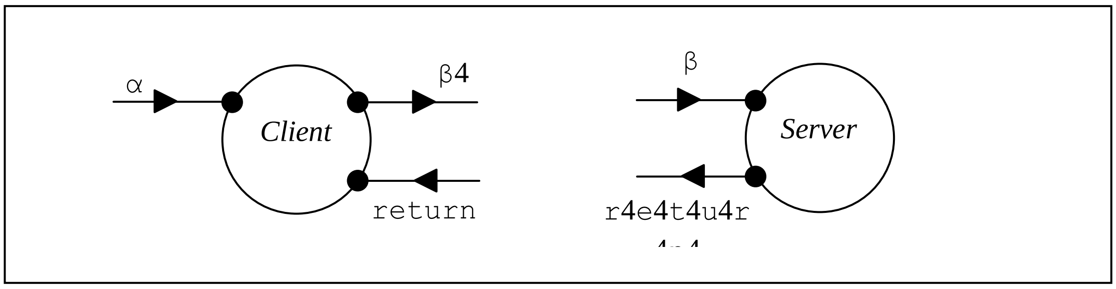

Figure 8. Client and Server in CCS Port diagrams - before linking

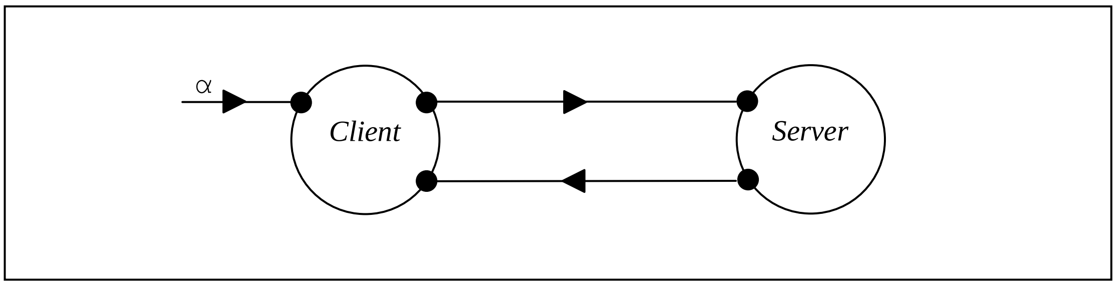

Figure 9. Client and Server in CCS Port diagrams - after linking

This composition can be defined by :
Client(Server 9 (Client[mid1/return, mid2/β4] | Server[mid1/ r3e4t4u4r4n4, mid2/β)\{mid1, mid2}

We see that a fired event on a transition becomes an output event in a CCS model. Where CCS restricts events, STATECRUNCHER allows for them to be labeled as *inter-component* (i.e. internal after composition) by means of a PCO. In this way, a test generator (or *Primer*), when communicating with STATECRUNCHER, can be instructed whether or not to exercise these events. Event α would be on a global PCO, or at on a PCO denoting a higher level of component aggregation.

CCS allows for replacement of simultaneous complementary events by τ, the "perfect action" [Milner p.39]. In our model, the transitions on β and β4 would be replaced by τ. When α takes place, τ *must* follow; nothing external can intervene (as it would spoil the paradigm). This is in accordance with CCS semantics, for although τ can lead to nondeterminism in an expression with a leading τ term such as

A9 $\alpha.A + \tau.b.A$ [Milner, p.42]

it is nevertheless permissible to eliminate τ when preceded by another event:

$\alpha.\tau.P = \alpha.P$ [Milner, p.41]

so we can be sure that τ takes place in our composition after event α.

By analogy with CCS, the STATECRUNCHER's `fire β` and transition in response to β are as good as simultaneous. This is a fair way to view STATECRUNCHER, since the transition semantics do not allow an intervening event. So we see a close parallel with CCS's notion of synchronization.

What if there are several clients?

Other clients of *Server* can also exist, but not be used simultaneously if there is just one instantiation of the server. Simultaneous outstanding server calls require the recursive state machine techniques of [StCrFunMod]. But provided the server is used sequentially, a STATECRUNCHER construction such as the following is useful:

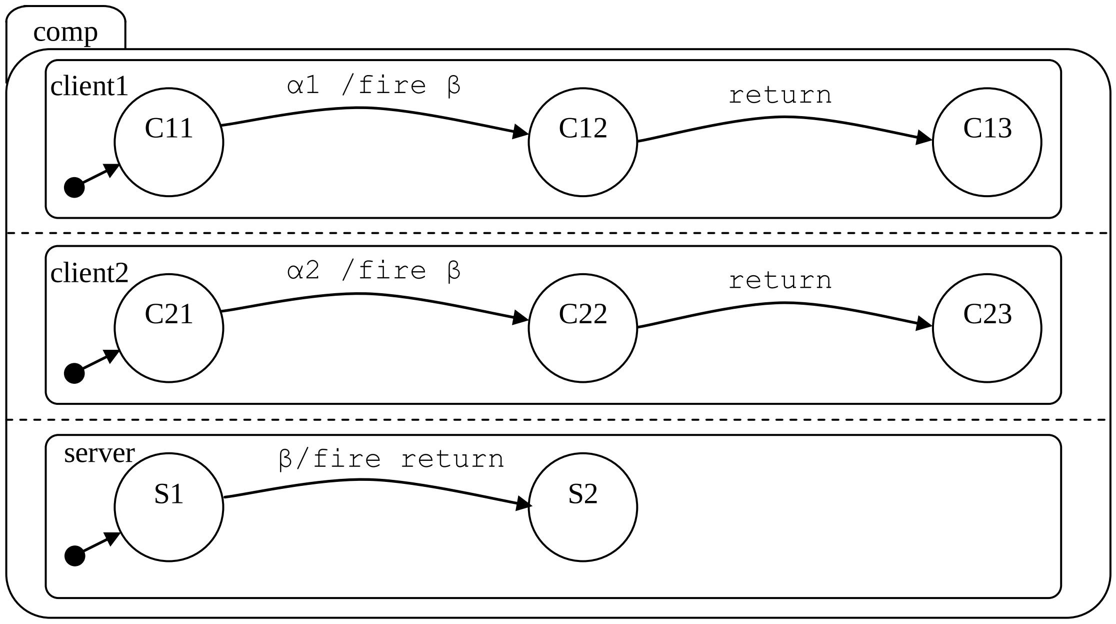

Figure 10. Several clients

In CCS, this can be modeled as follows:

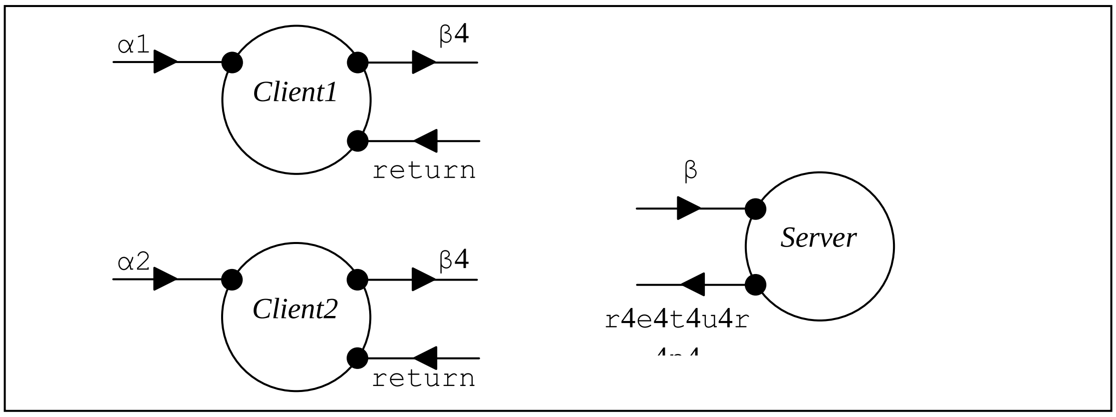

Figure 11. Multiple Clients in CCS - Port diagram

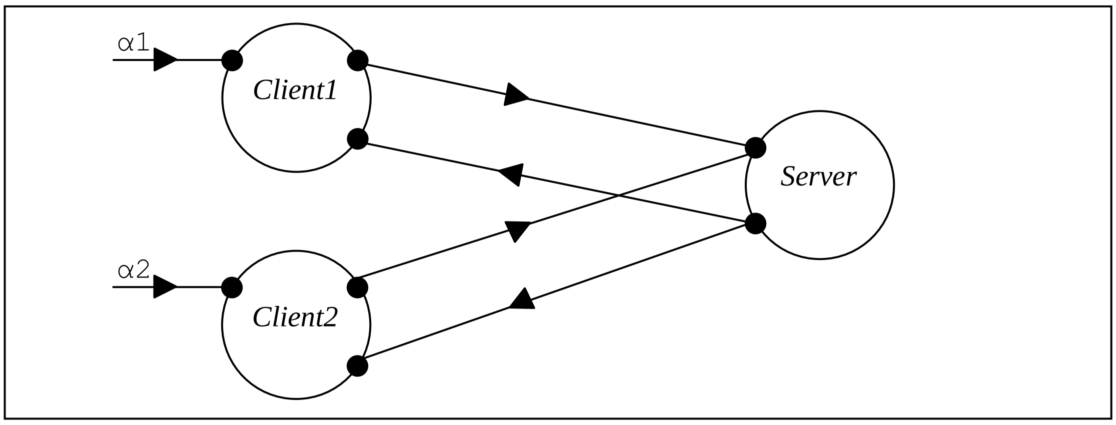

Figure 12. Multiple Clients composed in CCS - Port diagram

In this case we have the CCS composition

(Client1 | Client2 | Server)\{β,`return`}

The *Server* can synchronize with either client, as in the single client case. The clients never synchronize with each other.

What if there are several servers?

A server typically represents a 'C'-like function, and functions have unique names, and it is this name, β say, that will be in the `fire` β construction in STATECRUNCHER. So it is unlikely that the composition construction will be used with several servers - the system is rather nonsensical. Were this to be the case, however, the `fire` β action would synchronize with all servers. This cannot be modeled directly in CCS, as only two processes can synchronize. The `fire return` construction on return from the servers would be performed redundantly from all but the first server. This configuration does not appear to have any practical application.

CCS state view

The model of Figure 2 can be represented as follows in CCS, but we introduce additional states between the execution of α and the firing of β.

<u>Client</u>

C1 9 α.C1X
C1X 9 β3.C2
C2 9 `return`.C3

<u>Server</u>

S1 9 β.S1X
S1X 9 r4e4t4u4r4n4.S2

<u>The composition</u> restricts on β and `return`: (and so also on their complements)

COMP 9 C1 | S1 \{β,`return`}

The composition is shown in the figure below.

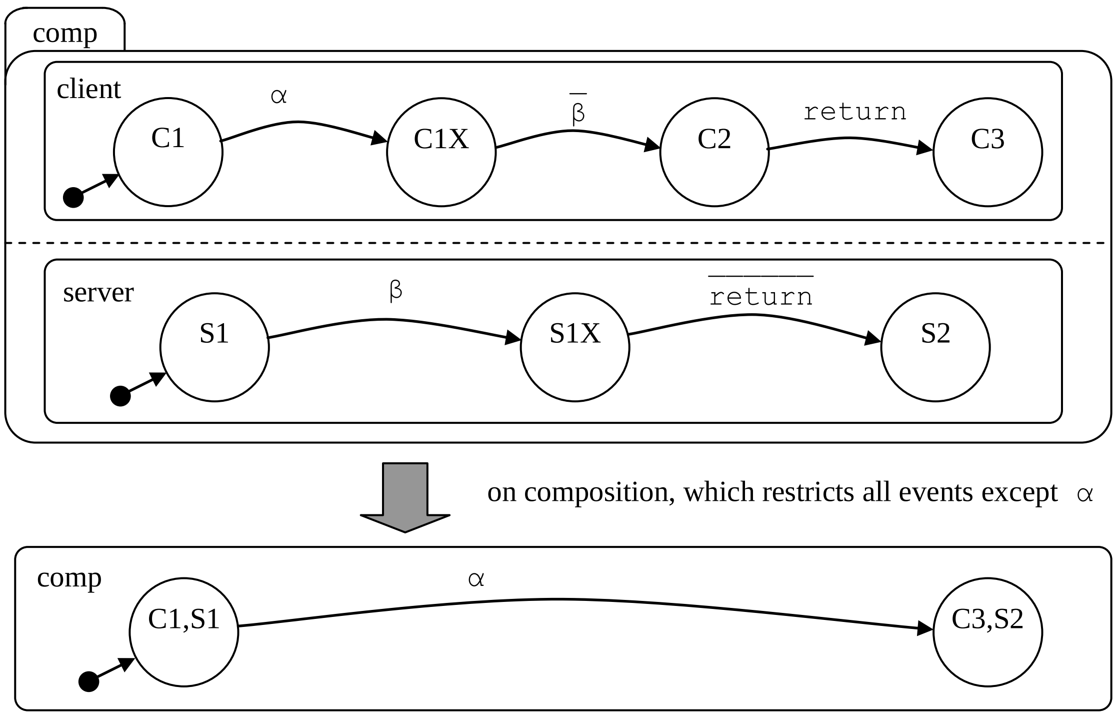
comp
client
C1
α
C1X
$\overline{\beta}$
C2
return
C3
server
S1
β
S1X
$\overline{\text{return}}$
S2
on composition, which restricts all events except α
comp
C1,S1
α
C3,S2

Figure 13. CCS state transition diagram of client-server model

Internal τ events arising from β/β3 and `return`/r4e4t4u4r4n4 events cause transitioning across states C1X, C2 and S1X, making them unobservable externally.

Modeling the interaction in CSP

The processes that engage are similar to those of CCS, but without complementary actions. If the processes had to share the same alphabet, we would compose with *C1X*||*S1* in the diagram below. The complete composition would be a process α^{a}(*C1X*||*S1*). But with the generalized composition operation, we can compose with *C1*||*S1*. The generalized composition operator would be essential if the client or server had additional states with their own events as in Figure 6 and Figure 7.

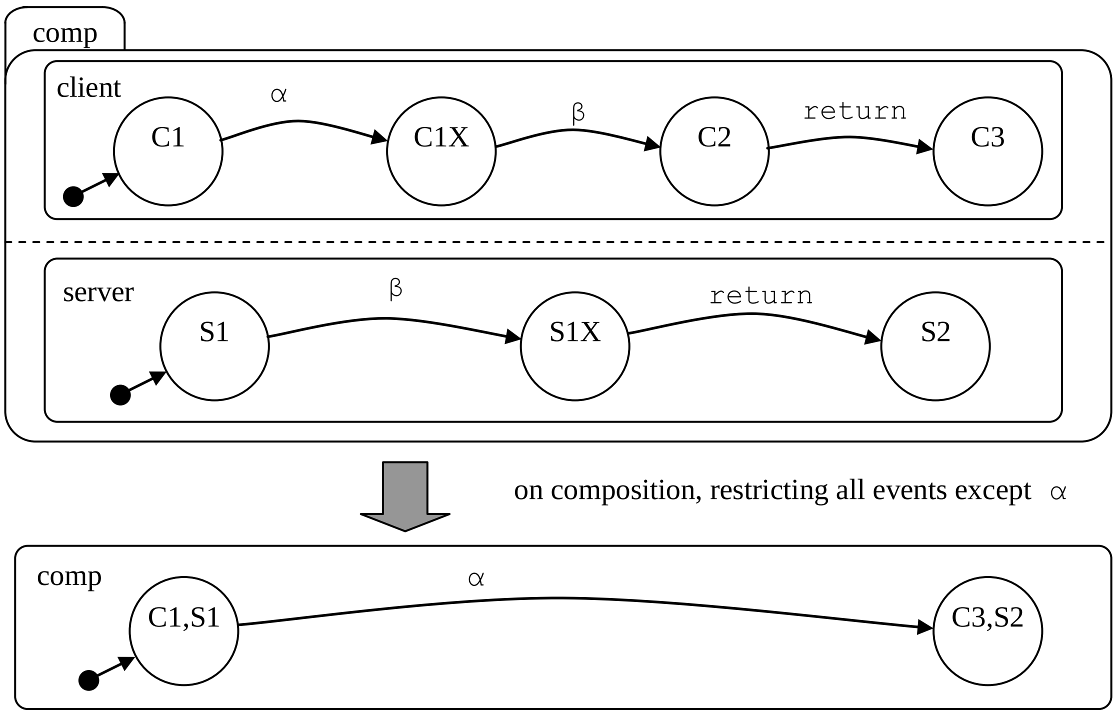

Figure 14. CSP state transition diagram of client-server model

The main differences in the approaches to composition are:

In CCS:

- Only 2 processes can participate in an interaction. They do this with complementary actions, which can be internalized into the internal event τ.
- The internal event τ may or may not take place and so gives rise to nondeterminism.

In CSP:

- There is no distinction between an event and its complement. Using the generalized composition operator as discussed, any number of processes with at least one some common event can be composed, but then all *must* participate in any such common event.

- So if one component of a composition is not at some stage able to respond to an event for the interaction, it will prevent the interaction.
- There is no τ nondeterminism.

In STATECRUNCHER:

- There is no symmetry between and `fire` β and β.
    - There can be several places where β is generated (including the user).
    - There can be several transitions triggered by β.
    - Some of these may have nothing to do with the composition. However, the event name would typically be reserved for the composition. It could be put on an inter-component PCO as a means of indicating that it is not available for independent generation.
- When β is generated, all transitions triggered will in principle take place, though they can be invalidated if at execution time due to preceding actions if their source state has been vacated or their guard condition has become false.

3. Parallelism other than call/return composition

The following examples show some situations that can arise with parallel systems, where the separate machines may influence the other's behaviour in a way other than an engagement in the sense considered in the previous section. We discuss them from the STATECRUNCHER perspective.

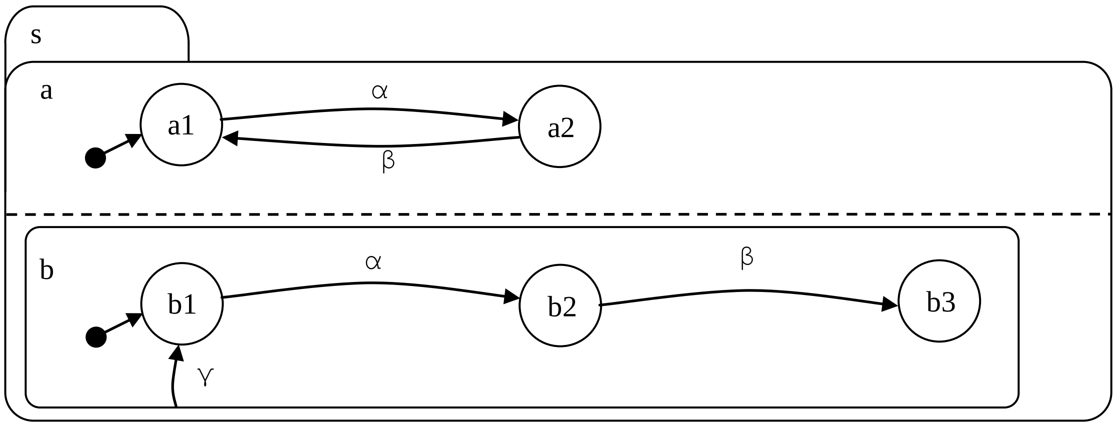

Figure 15. Simple parallelism

This machine represents a composition of clusters *a* and *b* in parallel. Cluster b is drawn with its own cluster boundary for clarity as to the tail of the transition on γ. In the initial state, in states *a1* and *b1*, the machine can respond to event α. The result will be that the clusters are in states *a2* and *b2*. This model may be appropriate under some circumstances, but there has been *no notion of interaction or synchronisation*.

Event α is not always processed in both clusters. If after first processing event α, we proceed to process event γ, the occupancy configuration is {a2,b1}. Now event α will only cause a transition in cluster *b*.

In fact, STATECRUNCHER interprets the above model as a race. Interleavings will be created. If we add some STATECRUNCHER actions to the transitions, this becomes apparent.

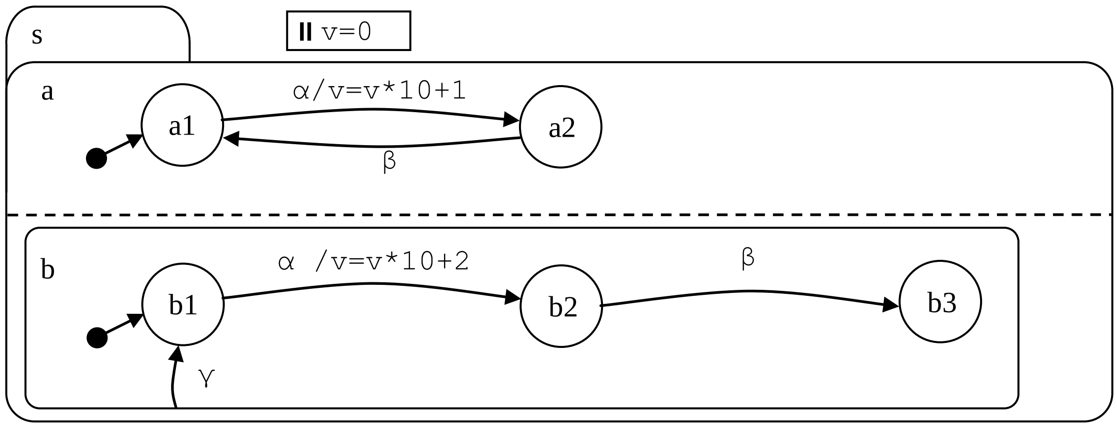

Figure 16. Simple parallelism as a race

In Figure 16, there is a variable `v` initialised to 0. The transition on α from a1 causes a digit 1 to be appended to the value of `v`. The transition on α from state b1 causes a digit 2 to be appended to the value of `v`. The result is 12 or 21 depending on the interleaving, i.e. who wins the race. STATECRUNCHER's nondeterminism handling produces a set of results, and so produces both values. Although we call this race nondeterminism, it is equivalent to fork nondeterminism in a flattened state space:

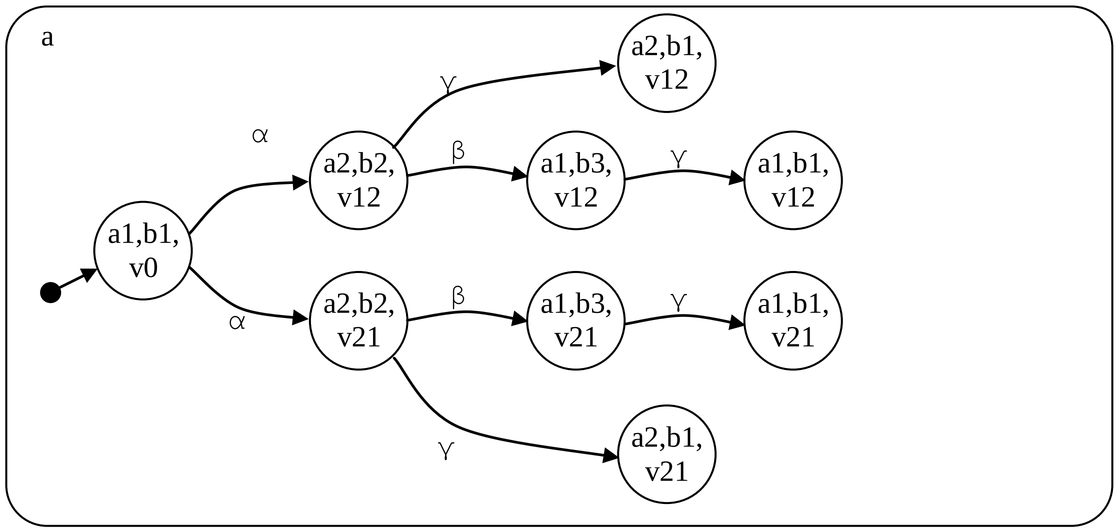

Figure 17. Part of flattened state space of the race model

The states are the Cartesian product of state occupancies in members `a` and `b` and with the values of variable `v`. The nondeterminism on event α from the initial state is seen as a fork.

The decision to evaluate conditions (guards) on transitions prior to executing them can lead to a blocked start, as in the following model.

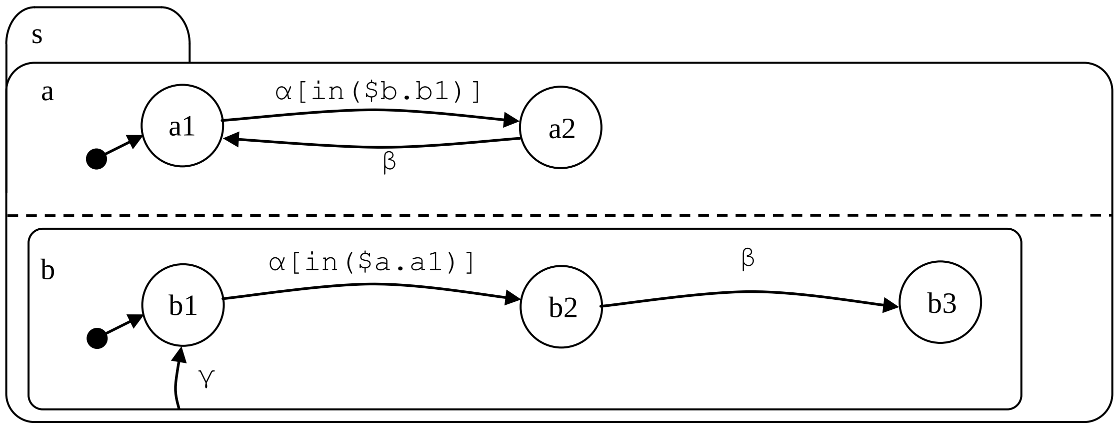

Figure 18. Blocked start

Each transition on α has a condition that the other cluster must be in its default state. So it appears that from the initial state of the composition, both transitions can take place. But since the transitions are executed sequentially, one will invalidate the other. STATECRUNCHER will produce two outcomes, one in {a1,b2} and one in {a2,b1}. The reasons for the choice of semantics are explored in the main thesis, but we give another example here showing why transitions cannot just be started in parallel:

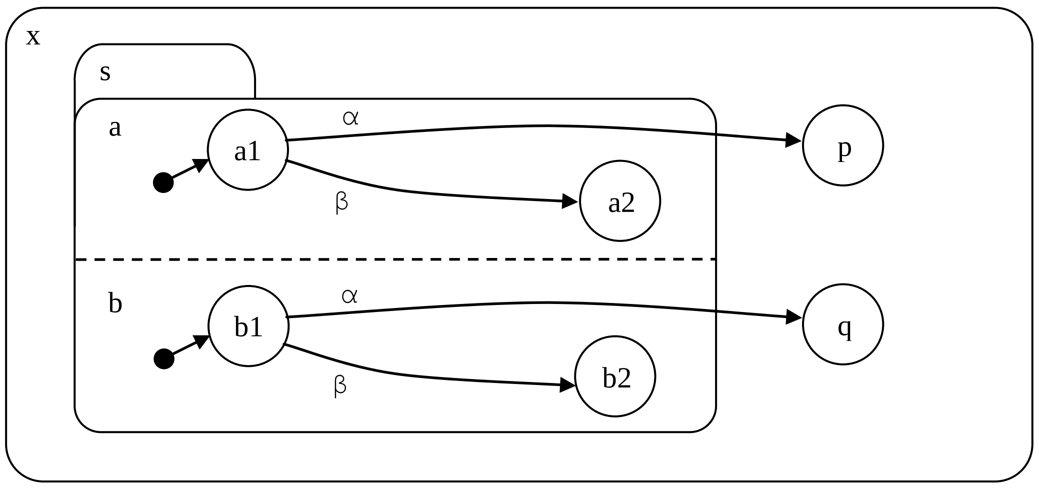

Figure 19. Parallel start problem

In this model, it appears that the two transitions on α can take place in parallel, but their target states are in conflict. They are members of the same cluster, and so cannot both become occupied. STATECRUNCHER's semantics are that after one transition, all conditions on the next are re-evaluated. The result is that STATECRUNCHER's two interleavings give a world in state *p* and a world in state *q* (set *s* is exited completely on either transition).

4. Nondeterminism

Nondeterminism in STATECRUNCHER

Finite state machines (FSMs) are often described without reference to the hierarchical structures of a UML or STATECRUNCHER statechart (in UML: concurrent and non-concurrent composite states; in STATECRUNCHER: sets and clusters). This is because the hierarchical structure is just a convenient way of expressing a mathematically equivalent flattened state space. When the hierarchy is introduced, the terminology changes from FSMs to *statecharts*, but the two are equivalent. A state in the flattened state space is an element of the Cartesian product of parallel states in the statechart. Only leafstates need be considered, because the occupancies of their ancestors is a derivative of that of the leafstates. If the statechart contains history, variables and traces, then these must also present as terms in the Cartesian product in defining flattened states.

An example has already been given in Figure 16 and Figure 17 where the effect of event α from the initial state is seen as race nondeterminism in the statechart and fork nondeterminism in the flattened state machine. The flattened state names are sequences (sequence brackets omitted for brevity). In the flattened state space, the only form of nondeterminism is fork nondeterminism.

From that example, it is seen that just as the hierarchical states of a *statechart* offer convenience in representing the *state space*, so some *nondeterministic semantics* (in this case, for the race) offer convenience in representing *FSM nondeterminism*. STATECRUNCHER simply structures the nondeterminism into various categories that are easy to visualize in a statechart.

STATECRUNCHER supports the following forms of structured nondeterminism:

- fork
- race
- set-transit
- set action
- set meta-event
- fired event (*or* broadcast event) nondeterminism

These are described in detail in [StCrMain]. After processing an event STATECRUNCHER produces a *world* per distinct state configuration, which, in flattened state space terms, is equivalent to a world for every possible resultant flattened state.

We develop the notion of a world more formally, working from the definition of a NFSM (Nondeterministic Finite State Machine) given by [Hierons]:

> An NFSM *M* is defined by a tuple *(S, s_1, h, X, Y)* in which
>
> - *S* is a set of states
> - s_1 is the initial state
> - *h* is the state transition function
> - *X* is the input alphabet
> - *Y* is the output alphabet
>
> Given an NFSM *M*, S_M shall denote the state set of *M*. When *M* receives an input value *x* ³ *X*, while in state *s* ³ *S*, a *transition* is executed producing an output value *y* ³ *Y* and moving *M* to some state *s'* ³ *S*. The function *h* gives the possible transitions and has the type $S \times X \rightarrow P(S \times Y)$ where *P* denotes the power set operator. ... An NFSM *M* is *completely specified* if, for each *s* ³ *S* and *x* ³ *X*, *|h(s,x)|*ä1. M is *deterministic* if for each *s* ³ *S* and *x* ³ *X*, $|h(s,x)| \leq 1$.

What in Hierons' description is the notion of *M* being *in state s*, is to STATECRUNCHER *having an occupancy configuration s, and other dynamic properties*, where an occupancy configuration gives the occupancy (occupied or vacant) of every state. Several states can be occupied, due to parallelism (modelled by a STATECRUNCHER *set*), and hierarchy (the fact that a parent of an occupied state is also an occupied state). Remark: the occupancy of non-leaf states can be derived from that of their child states (by the *set* and *cluster* rules), so, given the hierarchical structure, the occupancy configuration need only explicitly comprise the set of occupied leaf states.

The *'other dynamic properties'* which *s* must comprise are cluster history and variable values.

In our definitions below, we define G*(A×B)* ° P*(A×B)* to be the set of all *functions* from *A* to *B*.

A STATECRUNCHER statechart is therefore *(C, V, P, s_1, v_1, p_1, X, Y, h)* where

- *C* is a hierarchy of states (sets, clusters and leafstates), from which we can easily derive
  - *S*, the set of all states
  - *P*, the set of all clusters, *P* ¹ *S*
- *V* is a set of variables
- s_1 is the initial state
- v_1 is a function giving the initial variable values, $V \rightarrow Z$, where *Z* is the set of integers
- p_1 is a function giving the initial history values per cluster, $S \rightarrow S$
- *X* is the input alphabet (a set of events in STATECRUNCHER)
- *Y* is the output alphabet (a set of trace elements in STATECRUNCHER)
- *h* is the state transition function

  $$h : [S\times\ G(V\times Z) \times\ G(P\times S)]\times X\rightarrow P([S\times\ G(V\times Z) \times\ G(P\times S\ )]\times Y),$$ where
  - *the* G*(V×Z) term represents all the variables with their values*
  - *the* G*(P×S) term represents all the clusters with their histories*

◦ *the [...] bracketing on the LHS and RHS is introduced because of the commonality of these terms; they are the* STATECRUNCHER *worlds. There may be no worlds in existence.*

The domain and range of *h* can be represented as
domain (h) : [S × G(V×Z) × G(P×S)] × Y = W×X
range(h) :P([S × G(V×Z) × G(P×S)] × Y) = P(W×Y)

When an event is processed in many worlds, a new set of worlds is produced.
To represent this, we define a multi-input-world transition function
H: *P(W×X) → P(W×Y)*
H(A)=¿B³A h(B)

In a practical situation, the elements of the range of *H* will all contain the *same* event in all the Cartesian product terms.

Remark: in the actual STATECRUNCHER implementation, *traces* also distinguish worlds, so we should strictly say that dynamic the configuration *d* of a statechart is of type
*S × G(V×Z)× G(P×S)×Y**
where *Y** is the set of strings consisting of elements of *Y*, (including the empty sequence). So this could be considered to be the actual type of the range of the transition function *h*. However, the most efficient mode of operation is to clear traces and merge worlds between processing events; if this is not done, old and new traces are concatenated. Traces do not impinge on the transition algorithm. With this understanding, we discount the traces in a dynamic state, so we can more closely map to the description given by Hierons.

Comparison of nondeterminism

We take an example of fork nondeterminism:

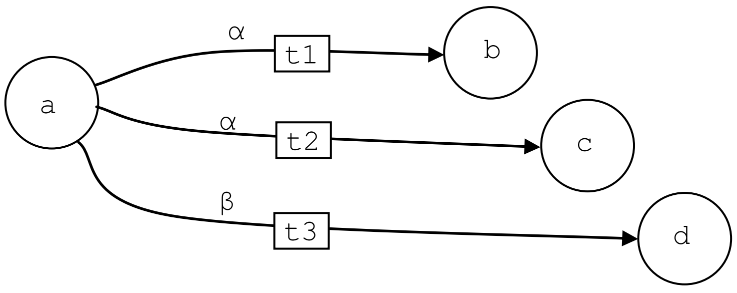

Figure 20. Simple fork nondeterminism

If state `a` is occupied, STATECRUNCHER offers the user a choice of transitionable events:

- event α
- event β

In STATECRUNCHER terminology, we say that event α leads to nondeterminism. STATECRUNCHER takes care of the nondeterministic outcomes without user interaction, by returning the set of all possible outcomes. If event α is selected, two *worlds* are produced, one in state `b` and one in state `c`, as described in [StCrMain]. In general there will be more than one world *beforehand* in which to process an event, and the event is processed in all of them.

Nondeterminism in CSP

In CSP, a choice between different *events* (e.g. α and β) is expressed by the *choice* operator (|) . So one process may be defined by

$(\alpha \to P \mid \beta \to Q)$.

This is not nondeterminism, since the environment can control such a process by the event given.

The choice operator (|) is not an operator on processes [Hoare, p31]. It is syntactically incorrect in CSP to write

$(\alpha \to P \mid \alpha \to Q)$

A process that behaves like P or Q where the environment has no control over the choice, is written using the *nondeterministic or* operator (n,), [Hoare p.102], which Schneider calls the *internal choice* operator [Schneider, p.24]. We can write

$(\alpha \to P)$ n $(\alpha \to Q)$.

CSP has another potentially nondeterministic operator, the *general choice* operator (o) [Hoare, p.106], which Schneider calls the *external choice* operator [Schneider, p.20]. The expression

(P o Q)

denotes a process which the environment can control, provided this is done by the first event. If only P can engage with the event, then P is selected. If only Q can engage with the event, then Q is selected. If both can engage with the event, then the choice is nondeterministic.

As mentioned previously, CSP has an *interleaving* operator |||, and in the expression $P|||Q$, if both processes can engage in an action, a nondeterministic choice is made between them. But unlike with (P o Q), no process is discarded, and the interleaving of two processes remains.

Compare again the ordinary CSP composition operator ||, whereby in $P||Q$, both processes must participate if the event is in both their alphabets.

We see that (P n Q), is nondeterministic, (P o Q) and (P ||| Q) can be nondeterministic, and (P || Q) is deterministic (inasmuch as P and Q are themselves deterministic).

Nondeterminism in CCS

CCS combines two agent expressions with the *summation* operator (+). This can be nondeterministic. From [Milner, p.20]:

> The agent *P+Q* behaves either like *P* or like *Q*; as soon as one performs its first action, the other is discarded. Often the environment will only permit one of these alternatives [...]. But if both alternatives are permitted, then *P+Q* is non-deterministic; that is, it may behave like *P* on one occasion and like *Q* on another.

CCS [Milner, p85] allows defining equations such as

B 9 *a.B1 + a.B1'*

where the same action occurs in more than one term on the right hand side.

CCS has additional nondeterminism on agent composition, because the internal transition τ may or may not occur [Milner p.40]. There can be several event-complement pairs that can give rise to different internal transitions. This means that several combinations of nondeterminism are possible:

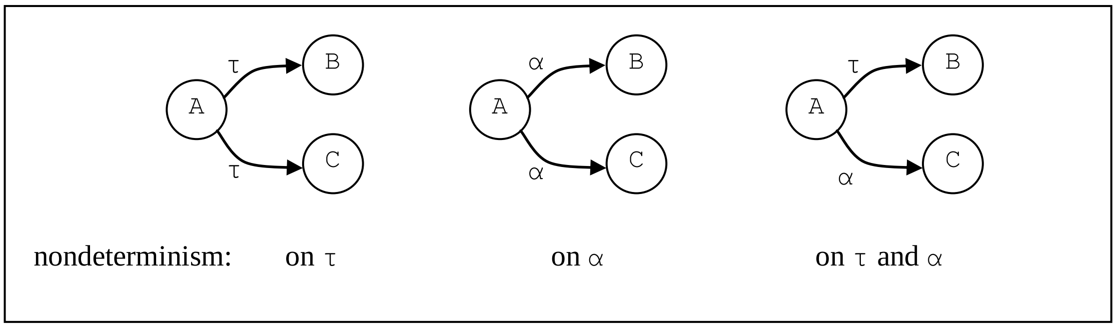

Figure 21. CCS combinations of nondeterminism

5. Concluding remarks

Concluding remarks on composition of processes

CCS is rather different to CSP and STATECRUNCHER, in that only two processes can participate in an interaction, but the event - event complement concept does match up with the STATECRUNCHER fired event mechanism for composition when there is one or more clients and one server. The CCS τ event may give rise to nondeterminism, where none would be present in the STATECRUNCHER composition paradigm as presented.

CSP does not have the two-process restriction of CCS, but there is no direct STATECRUNCHER counterpart to the way in which one participating process can *prevent* others from engaging (which happens in CSP when that process cannot respond to a particular event in the shared alphabet). Such a prevention mechanism is not required for simple client-server composition, (but constructs can be created as necessary - for a semaphore see the example of the dining philosophers in [StCrMain]).

Concluding remarks on nondeterminism

As mentioned, the forms of STATECRUNCHER nondeterminism (e.g. race nondeterminism), are simply convenient constructs for use in a structured way when dealing with a statechart structure containing hierarchy (clusters and sets) and concurrency (sets). These constructs are all effectively fork nondeterminism in an equivalent flattened model, and so are nothing new for the purposes of this comparison.

It is seen that the CCS *summation* operator (+) and the CSP *internal choice* operator (n) express nondeterminism in the STATECRUNCHER sense, but the operands must be processes not events, so the model of figure Figure 20 has to be expressed as separate processes rather than one process. This is effectively no more than a syntactic requirement of CCS and CSP. The semantics of CCS and CSP can lead to further nondeterministic situations, where a STATECRUNCHER model would typically contain a nondeterministic fork.

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

[Bruns]	Glenn Bruns Distributed Systems Analysis with CCS Prentice Hall 1997, ISBN 0-13-398389-7
[CHSM]	Paul J. Lucas An Object-Oriented System for Implementing Concurrent, Hierarchical, Finite State Machines. MSc. Thesis, University of Illinois at Urbana-Champaign, 1993
[Harel 87]	D. Harel *et al.* On the Formal Semantics of Statecharts Logic in Computer Science, 2nd Annual Conference, 1987, pp.54-64
[Hierons]	R.M. Hierons Adaptive testing of a deterministic implementation against a nondeterministic finite state machine. 1998
[Hoare]	C.A.R. Hoare Communicating Sequential Processes Prentice-Hall, 1985, ISBN 0-13-153271-5, 0-13-153289-8 PBK
[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.
[Milner]	Robin Milner Communication and Concurrency Prentice Hall 1997, ISBN 0-13-114984-9 and 0-13-115007-3 Pbk
[Schneider]	Steve Schneider Concurrent and Real-time Systems, The CSP Approach John Wiley & Sons Ltd, 2000, ISBN 0-471-62373-3

[Stannett]	Mike Stannett and A.J.H. Simons Complete Behavioural testing of Object-Oriented Systems using CCS-Augmented X-Machines Test Report CS-02-04, Dept. of Computer Science, United Kingdom
[TorX]	The *Côte de Résyste* Project delivers the TorX tool http://fmt.cs.utwente.nl/CdR
[UML]	The Object Management Group website is: http://www.omg.org UML specifications are available from this website.