Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison

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

In this paper we show how a system taken from the CCS literature can be modelled in STATECRUNCHER. An understanding of STATECRUNCHER is assumed, but for the purposes of this paper, most of STATECRUNCHER functionality will not seem strange to anyone familiar with UML dynamic modelling [UML], since that is the basis of the language.

We take a system that is neither too trivial nor too complex to serve as a good case study: the *distributed arbiter system* as described by Bruns [Bruns p.19]. For the definitive book on CCS by its designer, see [Milner].

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. Pre-requisite reading

Two separate appendices are pre-requisite reading to this appendix. They are:

- *A Semantic Comparison of STATECRUNCHER and Process Algebras* [StCrSemComp]
- *A Quick Reference of STATECRUNCHER's Output Format* [StCrOutput]

The first also describes differences in terminology between STATECRUNCHER and CCS, and compares their semantics and the way they compose separate state machines into a system. The nondeterministic features of STATECRUNCHER are discussed.

2. The distributed arbiter

2.1 Description of the problem

The purpose of this paper is to show how a system taken from the CCS literature can be modelled in STATECRUNCHER. We take a system that is neither too trivial nor too complex to serve as a good case study: The distributed arbiter system as described by Bruns [Bruns p.19]. For the definitive book on CCS by its designer, see [Milner].

The purpose of an arbiter is to manage a serially reusable resource, which we will henceforth just call the *resource*. If the resource is free, it can be allocated. If the resource is allocated, any other client has to wait (at least) until the resource is released.

Suppose there are two clients for a resource, and these clients run on separate machines in a network, and suppose that communication between the machines is regarded as expensive (probably in terms of time, affecting response times), so communication should be restricted to when it is essential. In this case, the combined arbiter can be constructed out of two single arbiters who share a token which gives the right to allocate the resource. Requests for the token, and replying to the requests by *passing the token* or saying *not-OK* will be only be performed if they are essential. So if a client on one machine obtains a resource, and then releases it and requests it again several times, without the other client requesting the resource, no traffic between the machines will ensue. For simplicity, the arbiter does not allow cancelling an unfulfilled request which has been placed for a resource - once a resource is requested, the client will either get it immediately or must wait for it. (That is how a normal program using a disk server will work, anyway). Figure 1 is a schematic of the distributed system.

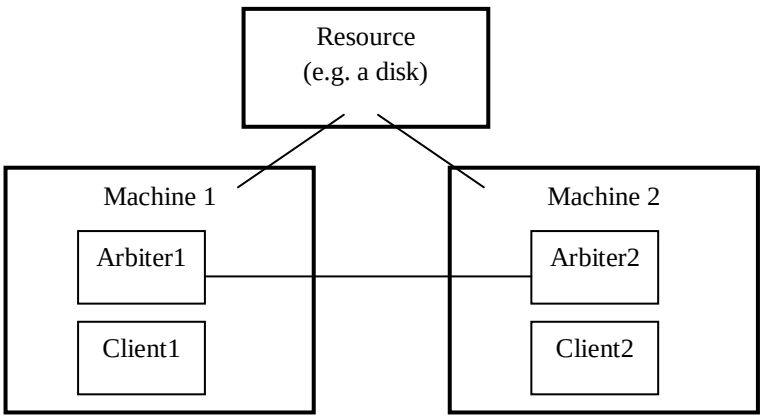


Figure 1. Distributed arbiters and clients for a resource

The CCS notation and state-transition diagram given for a single distributed arbiter are given in [Bruns, p.21] - but we first work in native STATECRUNCHER mode in designing a distributed arbiter. We return to Bruns's model after we have shown the STATECRUNCHER models.

We start with a single arbiter, instances of which run on each distributed machine. We call this arbiter in isolation **Me**, and its counterpart **You**, as a kind of template of the arbiter, but later in compositions we will name them John and Mary.

The following colour coding for events and PCOs (points of control and observation) will be used:

- **green** for a user (i.e. client) event that does not in itself interact with an arbiter, but which may be the event which makes the client want to interact with an arbiter.
- **blue** for client-arbiter interaction events corresponding to *requesting*, *releasing* and *acquiring* a resource from an arbiter. The *requesting* and *releasing* events are ones a user would supply; *acquiring* is one that would be supplied to a user.
- **red** for inter-arbiter events
- **black** for internal events to a single arbiter (if used)

Client-arbiter events: the client of the single arbiter can supply events

MeReqRes	Client tells Me-arbiter that it requests a resource
MeRelRes	Client tells Me-arbiter that it is releasing a resource
MeAcqRes	The Me-arbiter tells the client that the resource has been acquired

The PCO for these events is **ClientMePco**.

Inter-arbiter events: the events that occur between the arbiter and its counterpart are:

MeReqTok	I request the token
MePass	I pass the token to you
MeSayNok	I tell you you can't have the token

The PCO for these events is **InterArbMePco**.

YouReqTok	You request the token
YouPass	You pass the token to me
YouSayNok	You tell me I can't have the token

The PCO for these events is **InterArbYouPco**.

A STATECRUNCHER model of a single arbiter may or may not make use of internal events. That is perhaps a matter of taste. In this paper, we show two approaches, one with internal events and one without. If the first one seems unnecessary, when you come to it on page 7, skip it by going to page 16. To the outside world, which includes the other arbiter and the client, the behaviour is identical.

The following internal events are used in the relevant models, such as that of Figure 4. They are of local scope, and do not need a **Me** prefix.

TryTok I ask another part of myself whether I have the token, and if not, I tell that part of myself to try and get it. The responses may be **TryOk** (I already have the token), **YouSayNok** (after asking you for the token, I get a negative response), or **AcqTok** (I ask you for the token and you pass it to me).

TryOk I tell another part of myself that the try for the token succeeded (because I already had the token).

AcqTok I tell another part of myself that the try for the token succeeded (because I could get it from the You-arbiter).

ResetWant I tell another part of myself that the You-arbiter need no longer be considered wanting to obtain the token from me.

These events have a null PCO (denoted by [] in STATECRUNCHER).

The arbiter was initially modelled by the author with explicit internal parallelism for

- the state of resource allocation: Idle, Requested, Waiting, Alloc(ated)
- whether the arbiter possesses the token or not: NotHaveTok(en), HaveTok(en)
- whether the arbiter actually needs the token: NotNeedTok(en), NeedTok(en)
- whether the other arbiter wants the token: OtherNotWantTok(en), OtherWantTok(en).

We can flatten (explore) such a model. The Cartesian product of states is potentially 4x2x2x2 = 32, but actually only 6 can exist under proper sequences of events. The flattened model, which is without internal parallelism is presented later, and as in STATECRUNCHER it produces less output, we will mainly use it. The reader may regard the flattened model as more intuitive from the start or prefer the internally-parallel approach; we will first show the model with internal parallelism.

2.2 STATECRUNCHER notation and conventions

A few details of the model notation and STATECRUNCHER semantics are now explained for convenience. Details are available in the main STATECRUNCHER reports.

STATECRUNCHER's composite states are called *clusters* and *sets*. A cluster corresponds to a UML non-concurrent composite state, and to Harel's XOR-states [Harel]. A set corresponds to a UML concurrent composite state, and to Harel's AND-states.

STATECRUNCHER's "after-landing" transition semantics are essential. By this we mean that the actions associated with a transition are carried out after the target states of the transition has been entered, and the pure transition as such is complete. (The alternative is to carry out transition actions in "mid-flight"). In Figure 2, if we are in states *state1* and *state8*, and event `alpha` is given, the resultant states will be *state3* and *state9*. This is the basis of client-server communication modelling: when the client processes to event `alpha`, it needs to call the server, which is done by firing event `beta`. The return event from the server is event `gamma`.

The whole process is carried out without allowing other events to intervene. (Under mid-flight semantics, the transition on `gamma` would not take place, because its source state, *state2*, would not be occupied).

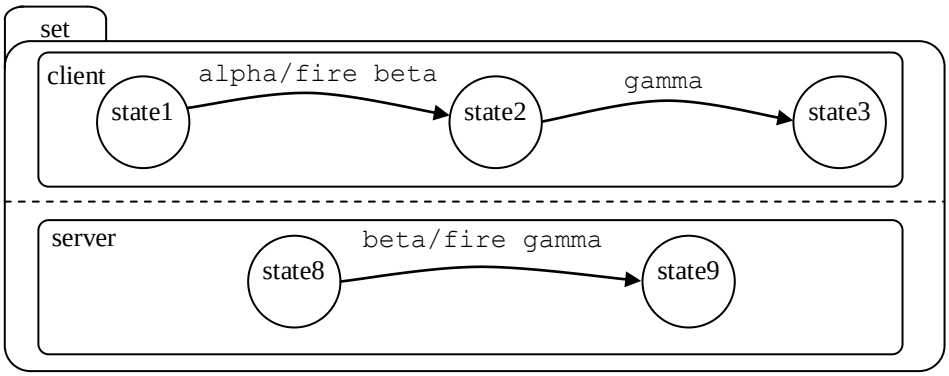


Figure 2. "After landing" semantics

STATECRUNCHER supports actions on transitions, as in Mealy machines, and actions on exiting or entering states (compare Moore machines). The only kind of action we are concerned with here is the *fired event*. (In other models, a variable assignment is a common action). The diagrammatic notation used here for on-entry actions is as follows with the arrow pointing into the state.

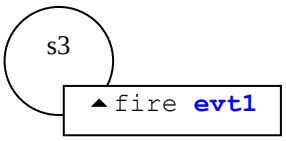


Figure 3. On entry symbol

Transitions can be triggered by internal STATECRUNCHER events, - the exiting and entering of states in a parallel part of the model. Such events are denoted by `enter(`*state*`)` and `exit(`*state*`)`.

Conditional *transitions* have a condition in square brackets. Conditional actions are represented by `if (`*condition*`)` *action* with optionally `else` *action*. Tests can be made for the occupancy of a parallel state using the `in(`*state*`)` function.

States are addressed using scoping operators. In brief, these are:

`$`*x*	go up a state then down into *x*
x`%%`*y*	go up until you reach *x*, then take *y*
a`.`*x*	descend through a and *x*
`::`*x*	start at statechart level and take *x*

They can be combined into expressions.

Traces in STATECRUNCHER (which are different in concept to those of CCS) are outputs that are observable at black-box level. They are generated in the model by the function `trace(`*expression*`)`, which can occur in any STATECRUNCHER action.

The commands to STATECRUNCHER that we will be using are

`?- cruncher.`	*enter the STATECRUNCHER read-process loop*
`SC:mm`	*set modelname mode*
`SC:run t4300`	*run a model (by modelname, not filename, in this mode)*
`SC:pe` *event*	*process an event*
`SC:gc`	*get configuration*
`SC:gt`	*get trace*

Reminder for users on how to exit the system

`SC:quit`	*quit the STATECRUNCHER read-process loop*
`?- halt.`	*exit Prolog*

2.3 The single arbiter

We now present the single arbiter, and show a session running it. Its STATECRUNCHER source, and that of other models, is given at the end of this paper.

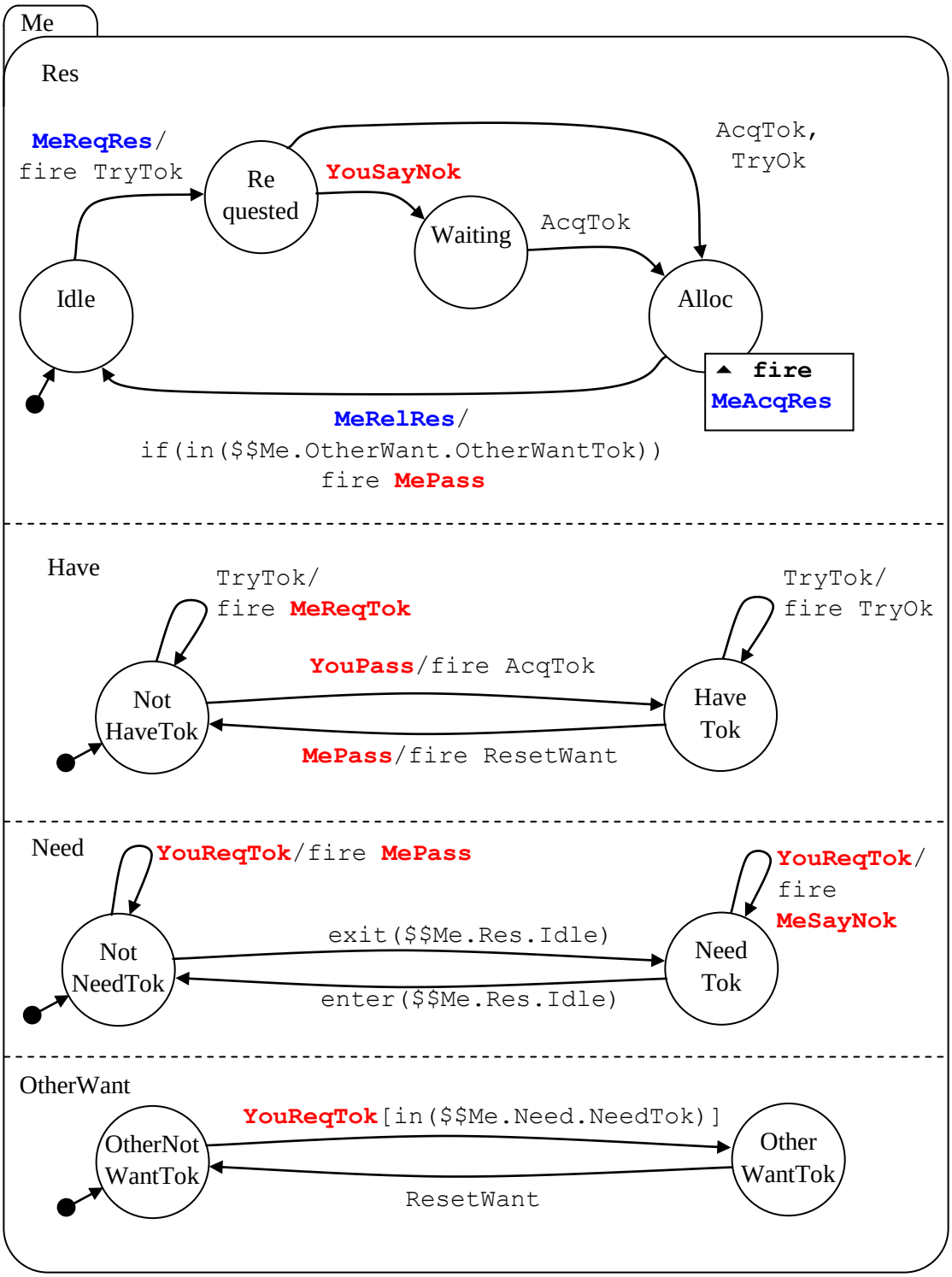


Figure 4. Single arbiter [model t4300]

Session with single arbiter `[model t4300]`

```
?- cruncher.
SC:mm
SC:run t4300
...
SC:gc
2    statechart sc
2       set Me [sc] = OCC []  **
2          cluster Res [Me, sc] = OCC []  **
2             leafstate Idle [Res, Me, sc] = OCC []  **
2             leafstate Requested [Res, Me, sc] = VAC []
2             leafstate Waiting [Res, Me, sc] = VAC []
2             leafstate Alloc [Res, Me, sc] = VAC []
2          cluster Have [Me, sc] = OCC []  **
2             leafstate NotHaveTok [Have, Me, sc] = OCC []  **
2             leafstate HaveTok [Have, Me, sc] = VAC []
2          cluster Need [Me, sc] = OCC []  **
2             leafstate NotNeedTok [Need, Me, sc] = OCC []  **
2             leafstate NeedTok [Need, Me, sc] = VAC []
2          cluster OtherWant [Me, sc] = OCC []  **
2             leafstate OtherNotWantTok [OtherWant, Me, sc] = OCC []  **
2             leafstate OtherWantTok [OtherWant, Me, sc] = VAC []
2    TRACE =[]
2    TREV [[MeReqRes, [sc]], 0, [], [ClientMePco, [sc]]]
2    TREV [[TryTok, [sc]], 0, [], []]
2    TREV [[YouPass, [sc]], 0, [], [InterArbYouPco, [sc]]]
2    TREV [[YouReqTok, [sc]], 0, [], [InterArbYouPco, [sc]]]

outworlds=[2]
number of outworlds=1
SC:pe MeReqRes
SC:gc
5    statechart sc
5       set Me [sc] = OCC []  **
5          cluster Res [Me, sc] = OCC []  **
5             leafstate Idle [Res, Me, sc] = VAC []
5             leafstate Requested [Res, Me, sc] = OCC []  **
5             leafstate Waiting [Res, Me, sc] = VAC []
5             leafstate Alloc [Res, Me, sc] = VAC []
5          cluster Have [Me, sc] = OCC []  **
5             leafstate NotHaveTok [Have, Me, sc] = OCC []  **
5             leafstate HaveTok [Have, Me, sc] = VAC []
5          cluster Need [Me, sc] = OCC []  **
5             leafstate NotNeedTok [Need, Me, sc] = VAC []
5             leafstate NeedTok [Need, Me, sc] = OCC []  **
5          cluster OtherWant [Me, sc] = OCC []  **
5             leafstate OtherNotWantTok [OtherWant, Me, sc] = OCC []  **
5             leafstate OtherWantTok [OtherWant, Me, sc] = VAC []
5    TRACE =[]
5    TREV [[AcqTok, [sc]], 0, [], []]
5    TREV [[TryOk, [sc]], 0, [], []]
5    TREV [[YouSayNok, [sc]], 0, [], [InterArbYouPco, [sc]]]
5    TREV [[TryTok, [sc]], 0, [], []]
5    TREV [[YouPass, [sc]], 0, [], [InterArbYouPco, [sc]]]
5    TREV [[YouReqTok, [sc]], 0, [], [InterArbYouPco, [sc]]]

outworlds=[5]
number of outworlds=1
SC:pe YouSayNok
SC:gc
6    statechart sc
6       set Me [sc] = OCC []  **
6          cluster Res [Me, sc] = OCC []  **
6             leafstate Idle [Res, Me, sc] = VAC []
6             leafstate Requested [Res, Me, sc] = VAC []
6             leafstate Waiting [Res, Me, sc] = OCC []  **
```



```
6             leafstate Alloc [Res, Me, sc] = VAC []
6          cluster Have [Me, sc] = OCC []  **
6             leafstate NotHaveTok [Have, Me, sc] = OCC []  **
6             leafstate HaveTok [Have, Me, sc] = VAC []
6          cluster Need [Me, sc] = OCC []  **
6             leafstate NotNeedTok [Need, Me, sc] = VAC []
6             leafstate NeedTok [Need, Me, sc] = OCC []  **
6          cluster OtherWant [Me, sc] = OCC []  **
6             leafstate OtherNotWantTok [OtherWant, Me, sc] = OCC []  **
6             leafstate OtherWantTok [OtherWant, Me, sc] = VAC []
6    TRACE =[]
6    TREV [[AcqTok, [sc]], 0, [], []]
6    TREV [[TryTok, [sc]], 0, [], []]
6    TREV [[YouPass, [sc]], 0, [], [InterArbYouPco, [sc]]]
6    TREV [[YouReqTok, [sc]], 0, [], [InterArbYouPco, [sc]]]

outworlds=[6]
number of outworlds=1
SC:pe YouPass
SC:gc
8    statechart sc
8       set Me [sc] = OCC []  **
8          cluster Res [Me, sc] = OCC []  **
8             leafstate Idle [Res, Me, sc] = VAC []
8             leafstate Requested [Res, Me, sc] = VAC []
8             leafstate Waiting [Res, Me, sc] = VAC []
8             leafstate Alloc [Res, Me, sc] = OCC []  **
8          cluster Have [Me, sc] = OCC []  **
8             leafstate NotHaveTok [Have, Me, sc] = VAC []
8             leafstate HaveTok [Have, Me, sc] = OCC []  **
8          cluster Need [Me, sc] = OCC []  **
8             leafstate NotNeedTok [Need, Me, sc] = VAC []
8             leafstate NeedTok [Need, Me, sc] = OCC []  **
8          cluster OtherWant [Me, sc] = OCC []  **
8             leafstate OtherNotWantTok [OtherWant, Me, sc] = OCC []  **
8             leafstate OtherWantTok [OtherWant, Me, sc] = VAC []
8    TRACE =[]
8    TREV [[MeRelRes, [sc]], 0, [], [ClientMePco, [sc]]]
8    TREV [[TryTok, [sc]], 0, [], []]
8    TREV [[MePass, [sc]], 0, [], [InterArbMePco, [sc]]]
8    TREV [[YouReqTok, [sc]], 0, [], [InterArbYouPco, [sc]]]

outworlds=[8]
number of outworlds=1
SC:
```

2.4 Two distributed arbiters

We now compose a system from two single arbiters, as shown in the following figure:

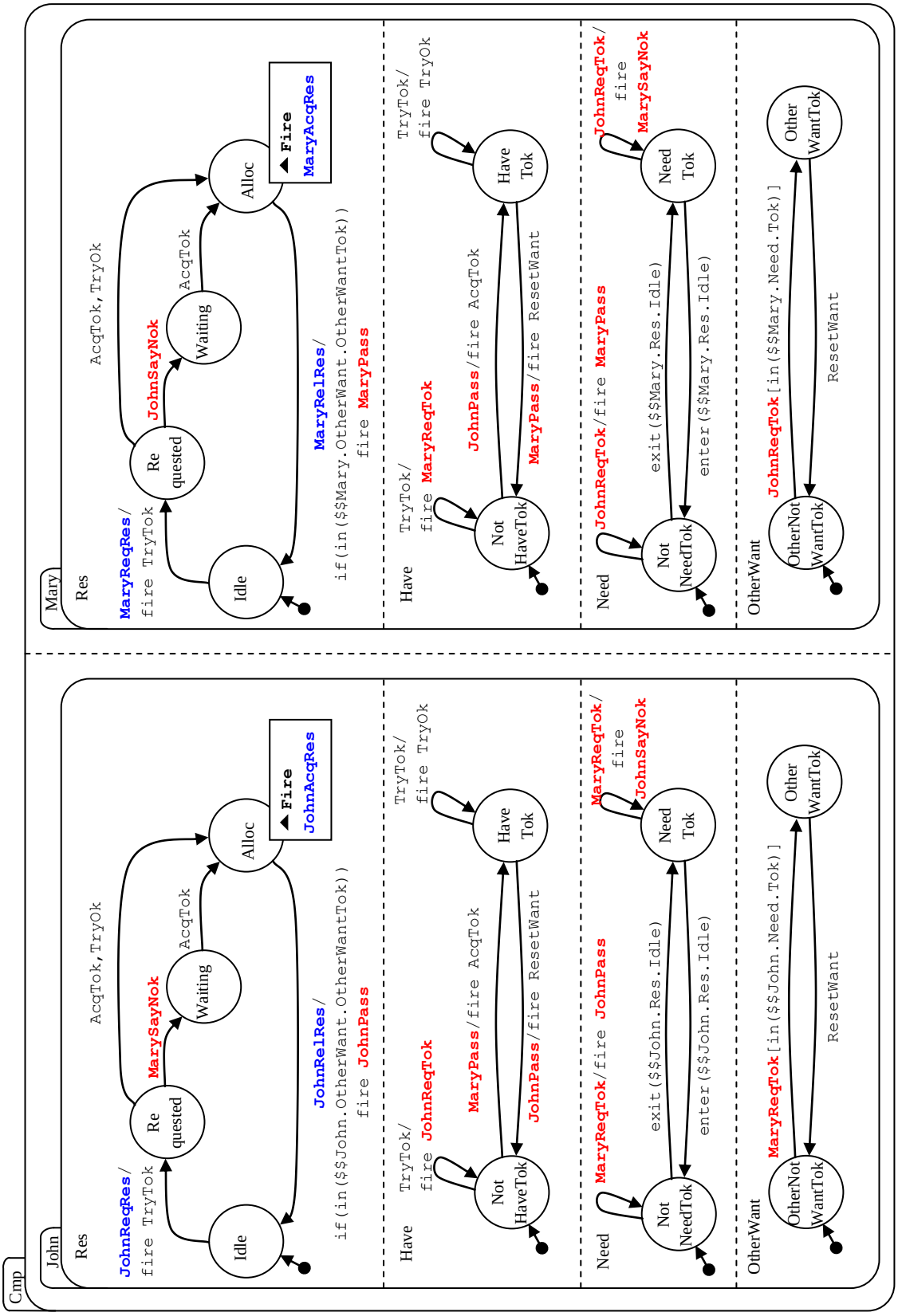


Figure 5. Two distributed arbiters (John and Mary) [model t4301]

Session with two distributed arbiters [model `t4301`]

Output not shown in full, as it is rather lengthy, and equivalent to that of model `t4311`, shown later.

```
?- cruncher.
SC:mm
SC:run t4301
...
SC:pe GiveJohnTok
SC:pe MaryReqRes
SC:pe JohnReqRes
SC:gc
19   statechart sc
19      set Cmp [sc] = OCC []  **
19         set John [Cmp, sc] = OCC []  **
19            cluster Res [John, Cmp, sc] = OCC []  **
19               leafstate Idle [Res, John, Cmp, sc] = VAC []
19               leafstate Requested [Res, John, Cmp, sc] = VAC []
19               leafstate Waiting [Res, John, Cmp, sc] = OCC []  **
19               leafstate Alloc [Res, John, Cmp, sc] = VAC []
19            cluster Have [John, Cmp, sc] = OCC []  **
19               leafstate NotHaveTok [Have, John, Cmp, sc] = OCC []  **
19               leafstate HaveTok [Have, John, Cmp, sc] = VAC []
19            cluster Need [John, Cmp, sc] = OCC []  **
19               leafstate NotNeedTok [Need, John, Cmp, sc] = VAC []
19               leafstate NeedTok [Need, John, Cmp, sc] = OCC []  **
19            cluster OtherWant [John, Cmp, sc] = OCC []  **
19               leafstate OtherNotWantTok [OtherWant, John, Cmp, sc] = OCC []  **
19               leafstate OtherWantTok [OtherWant, John, Cmp, sc] = VAC []
19         set Mary [Cmp, sc] = OCC []  **
19            cluster Res [Mary, Cmp, sc] = OCC []  **
19               leafstate Idle [Res, Mary, Cmp, sc] = VAC []
19               leafstate Requested [Res, Mary, Cmp, sc] = VAC []
19               leafstate Waiting [Res, Mary, Cmp, sc] = VAC []
19               leafstate Alloc [Res, Mary, Cmp, sc] = OCC []  **
19            cluster Have [Mary, Cmp, sc] = OCC []  **
19               leafstate NotHaveTok [Have, Mary, Cmp, sc] = VAC []
19               leafstate HaveTok [Have, Mary, Cmp, sc] = OCC []  **
19            cluster Need [Mary, Cmp, sc] = OCC []  **
19               leafstate NotNeedTok [Need, Mary, Cmp, sc] = VAC []
19               leafstate NeedTok [Need, Mary, Cmp, sc] = OCC []  **
19            cluster OtherWant [Mary, Cmp, sc] = OCC []  **
19               leafstate OtherNotWantTok [OtherWant, Mary, Cmp, sc] = VAC []
19               leafstate OtherWantTok [OtherWant, Mary, Cmp, sc] = OCC []  **
19   TRACE =[]
19   TREV [[AcqTok, [John, Cmp, sc]], 0, [], []]
19   TREV [[TryTok, [John, Cmp, sc]], 0, [], []]
19   TREV [[MaryPass, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
19   TREV [[MaryReqTok, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
19   TREV [[MaryRelRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
19   TREV [[TryTok, [Mary, Cmp, sc]], 0, [], []]
19   TREV [[JohnReqTok, [Cmp, sc]], 0, [], [InterArbJohnPco, [Cmp, sc]]]
19   TREV [[ResetWant, [Mary, Cmp, sc]], 0, [], []]
19   TREV [[GiveJohnTok, [sc]], 0, [], []]

outworlds=[19]
number of outworlds=1
SC:pe MaryRelRes
...
```

2.5 Two distributed arbiters with users

The following figure shows users (clients) composed into a system with two distributed arbiters:

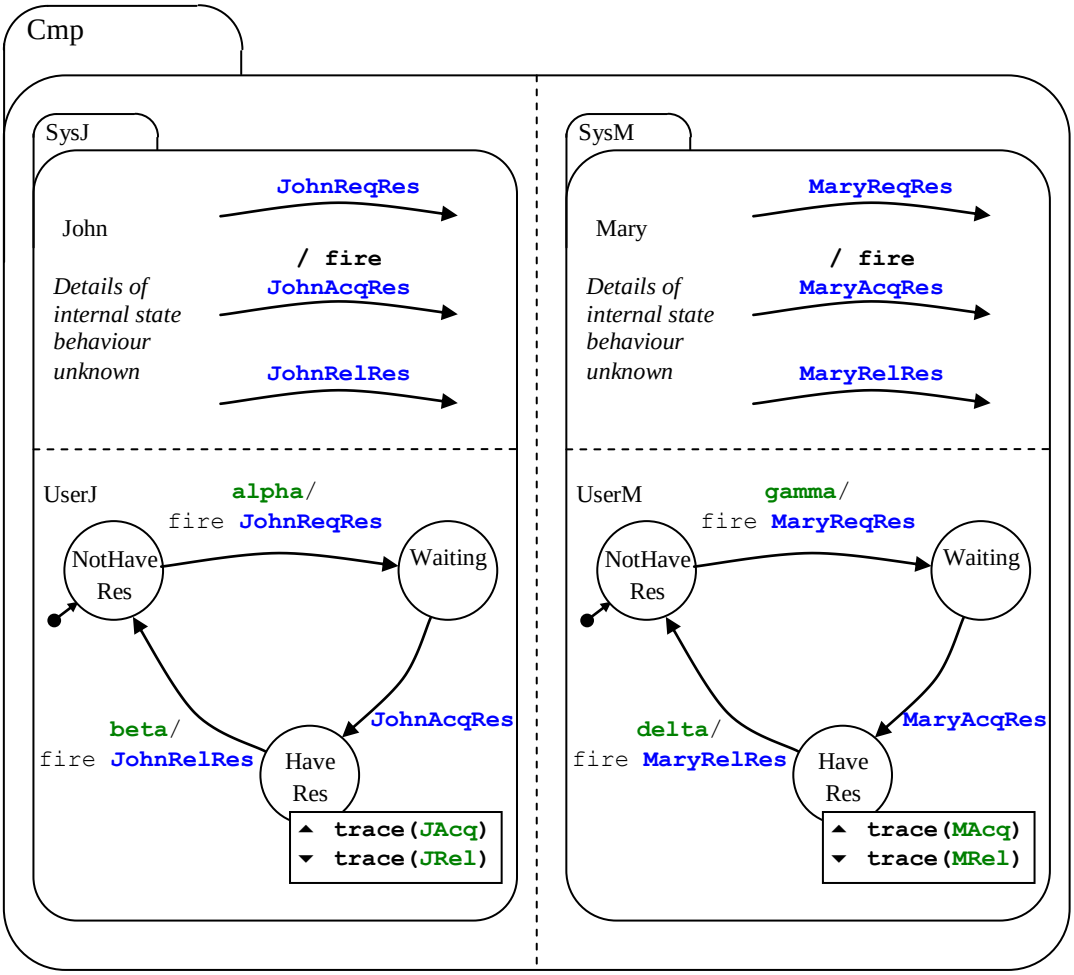


Figure 6. Two distributed arbiters with users (black box view) [model `t4302`]

In this model the clients need the resource when they process events **alpha** and **gamma**. The clients release the resource on events **beta** and **delta**. We drive the system using these events only.

Session with two distributed arbiters with users

Reminder: the `TRACE` is read from right to left.

Read

`JAcq` as *the user of arbiter John acquired the resource*
`JRel` as *the user of arbiter John released the resource*
`MAcq` as *the user of arbiter Mary acquired the resource*
`MRel` as *the user of arbiter Mary released the resource*

```
?- cruncher.
SC:mm
SC:run t4302
...
SC:gt
2    TRACE =[]
SC:pe GiveJohnTok
SC:gt
3    TRACE =[]
SC:pe alpha
SC:gt
10   TRACE =[JAcq]
SC:pe gamma
SC:gt
17   TRACE =[JAcq]
SC:pe beta
SC:gt
36   TRACE =[MAcq, JRel, JAcq]
SC:pe delta
SC:gt
40   TRACE =[MRel, MAcq, JRel, JAcq]


SC:gc
40   statechart sc
40      set Cmp [sc] = OCC []  **
40         set SysJ [Cmp, sc] = OCC []  **
40            set John [SysJ, Cmp, sc] = OCC []  **
40               cluster Res [John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate Idle [Res, John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate Requested [Res, John, SysJ, Cmp, sc] = VAC []
40                  leafstate Waiting [Res, John, SysJ, Cmp, sc] = VAC []
40                  leafstate Alloc [Res, John, SysJ, Cmp, sc] = VAC []
40               cluster Have [John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate NotHaveTok [Have, John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate HaveTok [Have, John, SysJ, Cmp, sc] = VAC []
40               cluster Need [John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate NotNeedTok [Need, John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate NeedTok [Need, John, SysJ, Cmp, sc] = VAC []
40               cluster OtherWant [John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate OtherNotWantTok [OtherWant, John, SysJ, Cmp, sc] = OCC []  **
40                  leafstate OtherWantTok [OtherWant, John, SysJ, Cmp, sc] = VAC []
40            cluster UserJ [SysJ, Cmp, sc] = OCC []  **
40               leafstate NotHaveRes [UserJ, SysJ, Cmp, sc] = OCC []  **
40               leafstate Waiting [UserJ, SysJ, Cmp, sc] = VAC []
40               leafstate HaveRes [UserJ, SysJ, Cmp, sc] = VAC []
40         set SysM [Cmp, sc] = OCC []  **
40            set Mary [SysM, Cmp, sc] = OCC []  **
40               cluster Res [Mary, SysM, Cmp, sc] = OCC []  **
40                  leafstate Idle [Res, Mary, SysM, Cmp, sc] = OCC []  **
40                  leafstate Requested [Res, Mary, SysM, Cmp, sc] = VAC []
40                  leafstate Waiting [Res, Mary, SysM, Cmp, sc] = VAC []
```

```
40                leafstate Alloc [Res, Mary, SysM, Cmp, sc] = VAC []
40             cluster Have [Mary, SysM, Cmp, sc] = OCC []  **
40                leafstate NotHaveTok [Have, Mary, SysM, Cmp, sc] = VAC []
40                leafstate HaveTok [Have, Mary, SysM, Cmp, sc] = OCC []  **
40             cluster Need [Mary, SysM, Cmp, sc] = OCC []  **
40                leafstate NotNeedTok [Need, Mary, SysM, Cmp, sc] = OCC []  **
40                leafstate NeedTok [Need, Mary, SysM, Cmp, sc] = VAC []
40             cluster OtherWant [Mary, SysM, Cmp, sc] = OCC []  **
40                leafstate OtherNotWantTok [OtherWant, Mary, SysM, Cmp, sc] = OCC []  **
40                leafstate OtherWantTok [OtherWant, Mary, SysM, Cmp, sc] = VAC []
40          cluster UserM [SysM, Cmp, sc] = OCC []  **
40             leafstate NotHaveRes [UserM, SysM, Cmp, sc] = OCC []  **
40             leafstate Waiting [UserM, SysM, Cmp, sc] = VAC []
40             leafstate HaveRes [UserM, SysM, Cmp, sc] = VAC []
40   TRACE =[MRel, MAcq, JRel, JAcq]
40   TREV [[JohnReqRes, [sc]], 0, [], [ClientJohnPco, [sc]]]
40   TREV [[TryTok, [John, SysJ, Cmp, sc]], 0, [], []]
40   TREV [[MaryPass, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
40   TREV [[MaryReqTok, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
40   TREV [[alpha, [sc]], 0, [], [UserJPco, [sc]]]
40   TREV [[MaryReqRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
40   TREV [[TryTok, [Mary, SysM, Cmp, sc]], 0, [], []]
40   TREV [[JohnReqTok, [Cmp, sc]], 0, [], [InterArbJohnPco, [Cmp, sc]]]
40   TREV [[gamma, [sc]], 0, [], [UserMPco, [sc]]]
40   TREV [[GiveJohnTok, [sc]], 0, [], []]

outworlds=[40]
number of outworlds=1
SC:
```

2.6 Flattened models

The diagrams following show an alternative model to the distributed arbiter, using just a cluster. We first show the flattened states of the model of Figure 4, then a new model based on the flattened states.

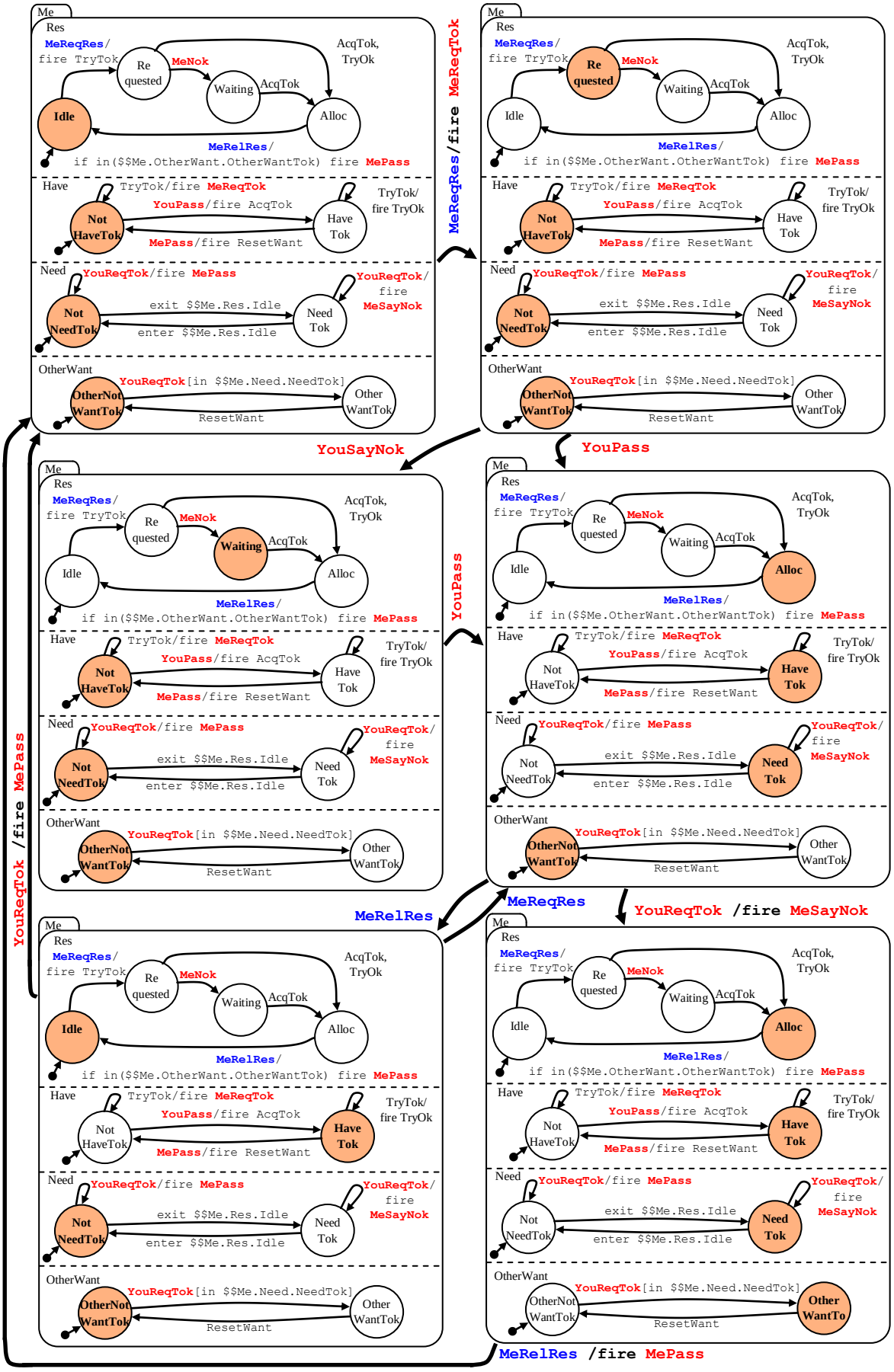


Figure 7. The model flattened (explored, unfolded)

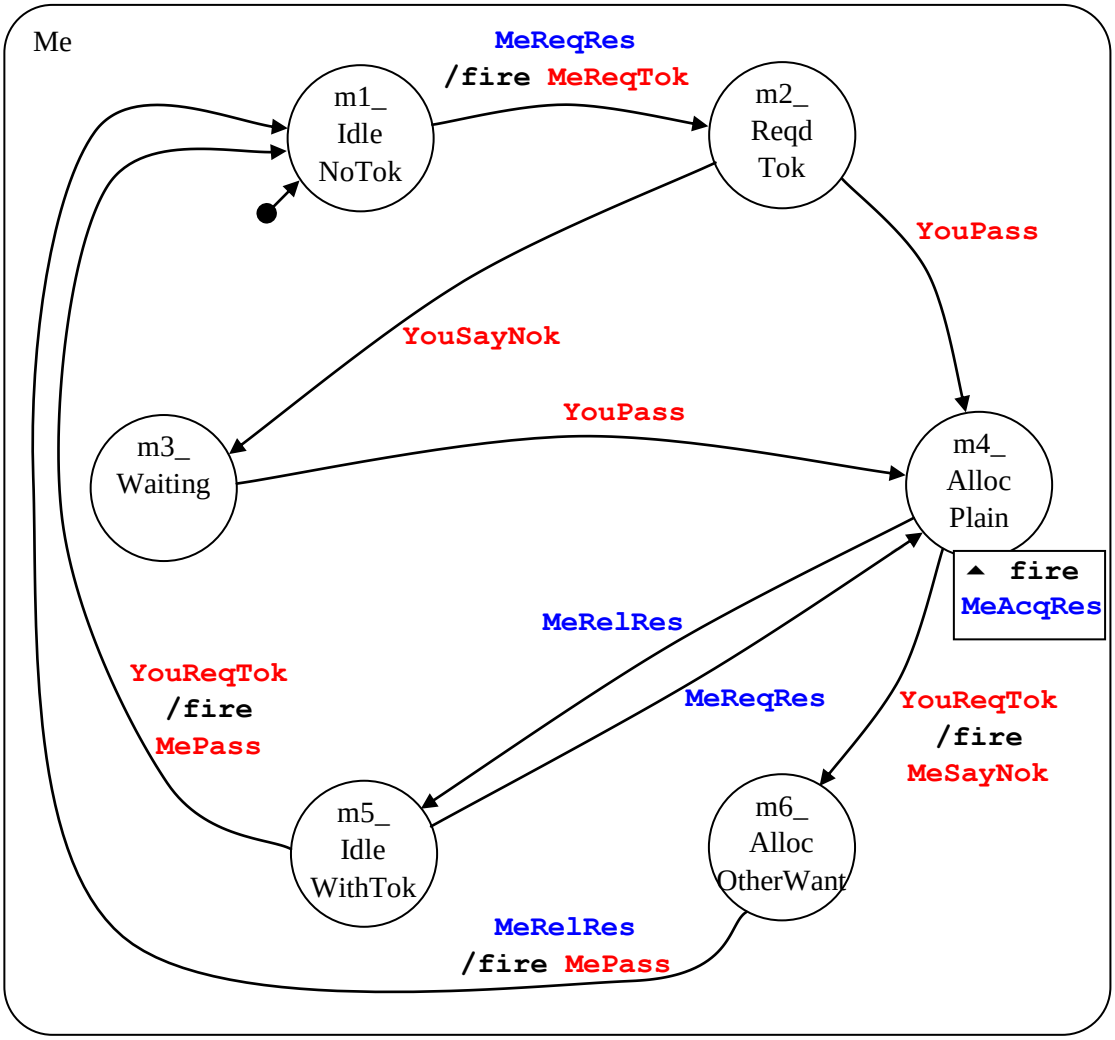


Figure 8. Single flattened distributed arbiter [model `t4310`]

Session with single flattened arbiter

```
| ?- cruncher.
SC:|: mm
SC:|: run t4310
...
SC:|: gc
2    statechart sc
2       cluster Me [sc] = OCC []  **
2          leafstate m1_IdleNoTok [Me,sc] = OCC []  **
2          leafstate m2_ReqdTok [Me,sc] = VAC []
2          leafstate m3_Waiting [Me,sc] = VAC []
2          leafstate m4_AllocPlain [Me,sc] = VAC []
2          leafstate m5_IdleWithTok [Me,sc] = VAC []
2          leafstate m6_AllocOtherWant [Me,sc] = VAC []
2    TRACE =[]
2    TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|: pe JohnReqRes
SC:|: gc
2    statechart sc
2       cluster Me [sc] = OCC []  **
2          leafstate m1_IdleNoTok [Me,sc] = OCC []  **
2          leafstate m2_ReqdTok [Me,sc] = VAC []
2          leafstate m3_Waiting [Me,sc] = VAC []
2          leafstate m4_AllocPlain [Me,sc] = VAC []
2          leafstate m5_IdleWithTok [Me,sc] = VAC []
2          leafstate m6_AllocOtherWant [Me,sc] = VAC []
2    TRACE =[]
2    TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|: pe MarySayNok
SC:|: gc
2    statechart sc
2       cluster Me [sc] = OCC []  **
2          leafstate m1_IdleNoTok [Me,sc] = OCC []  **
2          leafstate m2_ReqdTok [Me,sc] = VAC []
2          leafstate m3_Waiting [Me,sc] = VAC []
2          leafstate m4_AllocPlain [Me,sc] = VAC []
2          leafstate m5_IdleWithTok [Me,sc] = VAC []
2          leafstate m6_AllocOtherWant [Me,sc] = VAC []
2    TRACE =[]
2    TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|: pe MaryPass
SC:|: gc
2    statechart sc
2       cluster Me [sc] = OCC []  **
2          leafstate m1_IdleNoTok [Me,sc] = OCC []  **
2          leafstate m2_ReqdTok [Me,sc] = VAC []
2          leafstate m3_Waiting [Me,sc] = VAC []
2          leafstate m4_AllocPlain [Me,sc] = VAC []
2          leafstate m5_IdleWithTok [Me,sc] = VAC []
2          leafstate m6_AllocOtherWant [Me,sc] = VAC []
2    TRACE =[]
2    TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|:
```

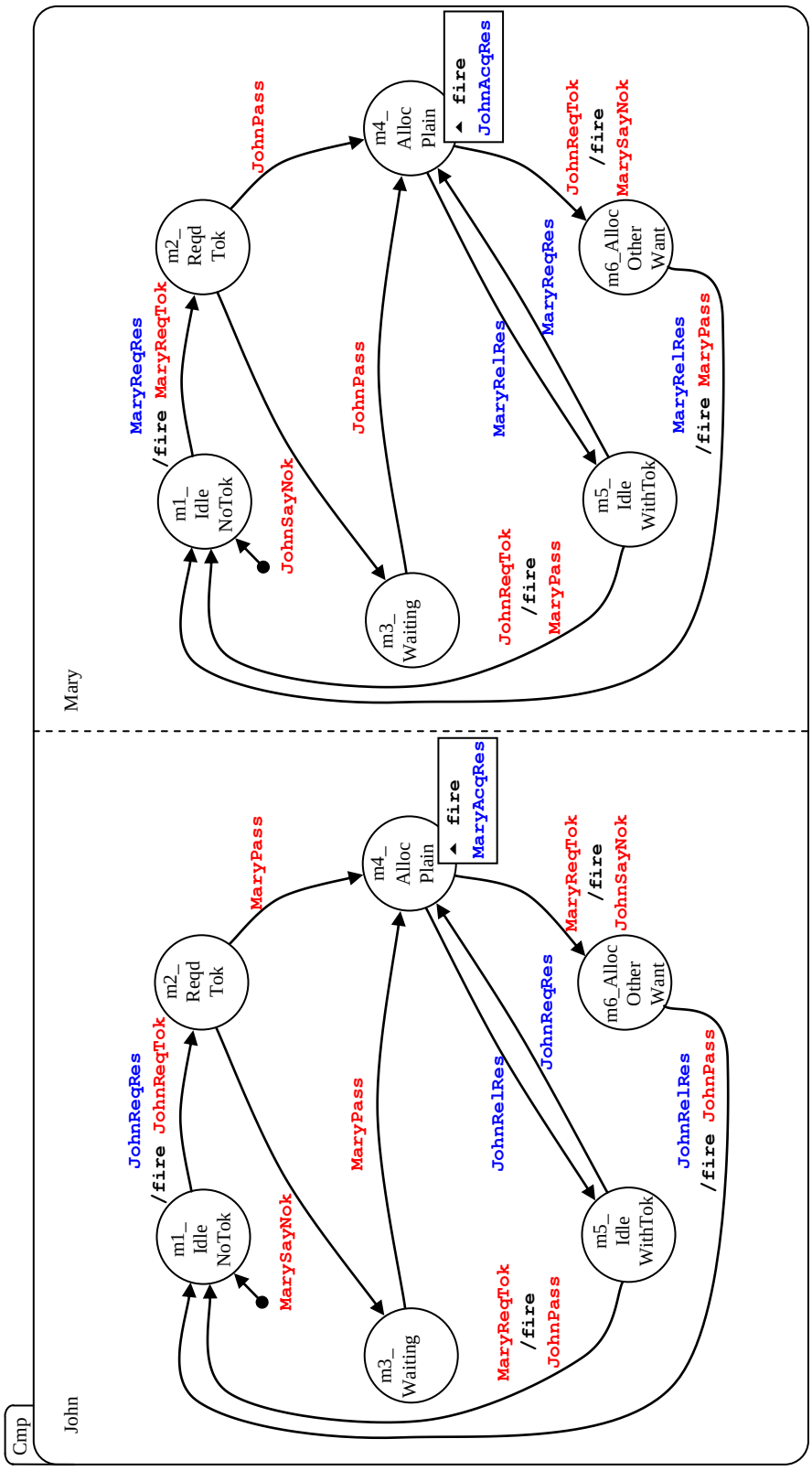
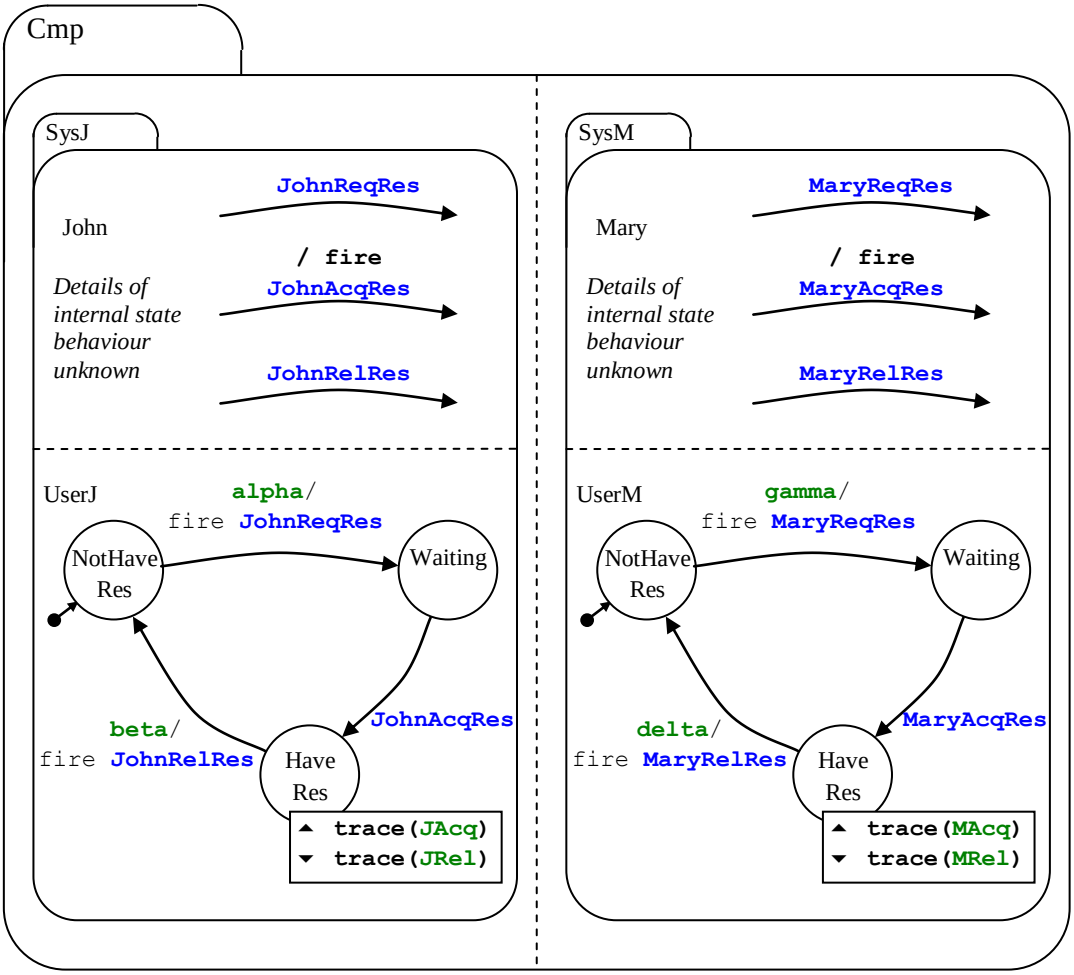


Figure 9. Two flattened distributed arbiters [model `t4311`]

Session with two flattened distributed arbiters

```
| ?- cruncher.
SC:|: mm
SC:|: run t4311
...
SC:|: pe GiveJohnTok
SC:|: gc
3    statechart sc
3       set Cmp [sc] = OCC []  **
3          cluster John [Cmp,sc] = OCC []  **
3             leafstate m1_IdleNoTok [John,Cmp,sc] = VAC []
3             leafstate m2_ReqdTok [John,Cmp,sc] = VAC []
3             leafstate m3_Waiting [John,Cmp,sc] = VAC []
3             leafstate m4_AllocPlain [John,Cmp,sc] = VAC []
3             leafstate m5_IdleWithTok [John,Cmp,sc] = OCC []  **
3             leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
3          cluster Mary [Cmp,sc] = OCC []  **
3             leafstate m1_IdleNoTok [Mary,Cmp,sc] = OCC []  **
3             leafstate m2_ReqdTok [Mary,Cmp,sc] = VAC []
3             leafstate m3_Waiting [Mary,Cmp,sc] = VAC []
3             leafstate m4_AllocPlain [Mary,Cmp,sc] = VAC []
3             leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
3             leafstate m6_AllocOtherWant [Mary,Cmp,sc] = VAC []
3    TRACE =[]
3    TREV [[MaryReqTok,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
3    TREV [[JohnReqRes,[sc]],0,[],[ClientJohnPco,[sc]]]
3    TREV [[MaryReqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
3    TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[3]
number of outworlds=1
SC:|: pe MaryReqRes
SC:|: gc
6    statechart sc
6       set Cmp [sc] = OCC []  **
6          cluster John [Cmp,sc] = OCC []  **
6             leafstate m1_IdleNoTok [John,Cmp,sc] = OCC []  **
6             leafstate m2_ReqdTok [John,Cmp,sc] = VAC []
6             leafstate m3_Waiting [John,Cmp,sc] = VAC []
6             leafstate m4_AllocPlain [John,Cmp,sc] = VAC []
6             leafstate m5_IdleWithTok [John,Cmp,sc] = VAC []
6             leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
6          cluster Mary [Cmp,sc] = OCC []  **
6             leafstate m1_IdleNoTok [Mary,Cmp,sc] = VAC []
6             leafstate m2_ReqdTok [Mary,Cmp,sc] = VAC []
6             leafstate m3_Waiting [Mary,Cmp,sc] = VAC []
6             leafstate m4_AllocPlain [Mary,Cmp,sc] = OCC []  **
6             leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
6             leafstate m6_AllocOtherWant [Mary,Cmp,sc] = VAC []
6    TRACE =[]
6    TREV [[JohnReqRes,[sc]],0,[],[ClientJohnPco,[sc]]]
6    TREV [[MaryRelRes,[sc]],0,[],[ClientMaryPco,[sc]]]
6    TREV [[JohnReqTok,[Cmp,sc]],0,[],[InterArbJohnPco,[Cmp,sc]]]
6    TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[6]
number of outworlds=1
SC:|: pe JohnReqRes
SC:|: gc
9    statechart sc
9       set Cmp [sc] = OCC []  **
9          cluster John [Cmp,sc] = OCC []  **
9             leafstate m1_IdleNoTok [John,Cmp,sc] = VAC []
9             leafstate m2_ReqdTok [John,Cmp,sc] = VAC []
9             leafstate m3_Waiting [John,Cmp,sc] = OCC []  **
9             leafstate m4_AllocPlain [John,Cmp,sc] = VAC []
```

```
9             leafstate m5_IdleWithTok [John,Cmp,sc] = VAC []
9             leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
9          cluster Mary [Cmp,sc] = OCC []  **
9             leafstate m1_IdleNoTok [Mary,Cmp,sc] = VAC []
9             leafstate m2_ReqdTok [Mary,Cmp,sc] = VAC []
9             leafstate m3_Waiting [Mary,Cmp,sc] = VAC []
9             leafstate m4_AllocPlain [Mary,Cmp,sc] = VAC []
9             leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
9             leafstate m6_AllocOtherWant [Mary,Cmp,sc] = OCC []  **
9    TRACE =[]
9    TREV [[MaryPass,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
9    TREV [[MaryRelRes,[sc]],0,[],[ClientMaryPco,[sc]]]
9    TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[9]
number of outworlds=1
SC:|: pe MaryRelRes
SC:|: gc
11   statechart sc
11      set Cmp [sc] = OCC []  **
11         cluster John [Cmp,sc] = OCC []  **
11            leafstate m1_IdleNoTok [John,Cmp,sc] = VAC []
11            leafstate m2_ReqdTok [John,Cmp,sc] = VAC []
11            leafstate m3_Waiting [John,Cmp,sc] = VAC []
11            leafstate m4_AllocPlain [John,Cmp,sc] = OCC []  **
11            leafstate m5_IdleWithTok [John,Cmp,sc] = VAC []
11            leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
11         cluster Mary [Cmp,sc] = OCC []  **
11            leafstate m1_IdleNoTok [Mary,Cmp,sc] = OCC []  **
11            leafstate m2_ReqdTok [Mary,Cmp,sc] = VAC []
11            leafstate m3_Waiting [Mary,Cmp,sc] = VAC []
11            leafstate m4_AllocPlain [Mary,Cmp,sc] = VAC []
11            leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
11            leafstate m6_AllocOtherWant [Mary,Cmp,sc] = VAC []
11   TRACE =[]
11   TREV [[JohnRelRes,[sc]],0,[],[ClientJohnPco,[sc]]]
11   TREV [[MaryReqTok,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
11   TREV [[MaryReqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
11   TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[11]
number of outworlds=1
SC:|:
```

Flattened model and session with users - **Repeat of Figure 6**, [model `t4312`]

```
| ?- cruncher.
SC:|: mm
SC:|: run t4312
...
SC:|: pe GiveJohnTok
SC:|: gc
3    statechart sc
3       set Cmp [sc] = OCC []  **
3          set SysJ [Cmp,sc] = OCC []  **
3             cluster John [SysJ,Cmp,sc] = OCC []  **
3                leafstate m1_IdleNoTok [John,SysJ,Cmp,sc] = VAC []
3                leafstate m2_ReqdTok [John,SysJ,Cmp,sc] = VAC []
3                leafstate m3_Waiting [John,SysJ,Cmp,sc] = VAC []
3                leafstate m4_AllocPlain [John,SysJ,Cmp,sc] = VAC []
3                leafstate m5_IdleWithTok [John,SysJ,Cmp,sc] = OCC []  **
3                leafstate m6_AllocOtherWant [John,SysJ,Cmp,sc] = VAC []
3             cluster UserJ [SysJ,Cmp,sc] = OCC []  **
3                leafstate NotHaveRes [UserJ,SysJ,Cmp,sc] = OCC []  **
3                leafstate Waiting [UserJ,SysJ,Cmp,sc] = VAC []
3                leafstate HaveRes [UserJ,SysJ,Cmp,sc] = VAC []
3          set SysM [Cmp,sc] = OCC []  **
3             cluster Mary [SysM,Cmp,sc] = OCC []  **
3                leafstate m1_IdleNoTok [Mary,SysM,Cmp,sc] = OCC []  **
```

```
3                leafstate m2_ReqdTok [Mary,SysM,Cmp,sc] = VAC []
3                leafstate m3_Waiting [Mary,SysM,Cmp,sc] = VAC []
3                leafstate m4_AllocPlain [Mary,SysM,Cmp,sc] = VAC []
3                leafstate m5_IdleWithTok [Mary,SysM,Cmp,sc] = VAC []
3                leafstate m6_AllocOtherWant [Mary,SysM,Cmp,sc] = VAC []
3             cluster UserM [SysM,Cmp,sc] = OCC []  **
3                leafstate NotHaveRes [UserM,SysM,Cmp,sc] = OCC []  **
3                leafstate Waiting [UserM,SysM,Cmp,sc] = VAC []
3                leafstate HaveRes [UserM,SysM,Cmp,sc] = VAC []
3    TRACE =[]
3    TREV [[MaryReqTok,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
3    TREV [[JohnReqRes,[sc]],0,[],[ClientJohnPco,[sc]]]
3    TREV [[alpha,[sc]],0,[],[UserJPco,[sc]]]
3    TREV [[MaryReqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
3    TREV [[gamma,[sc]],0,[],[UserMPco,[sc]]]
3    TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[3]
number of outworlds=1
SC:|: pe alpha
SC:|: gc
7    statechart sc
7       set Cmp [sc] = OCC []  **
7          set SysJ [Cmp,sc] = OCC []  **
7             cluster John [SysJ,Cmp,sc] = OCC []  **
7                leafstate m1_IdleNoTok [John,SysJ,Cmp,sc] = VAC []
7                leafstate m2_ReqdTok [John,SysJ,Cmp,sc] = VAC []
7                leafstate m3_Waiting [John,SysJ,Cmp,sc] = VAC []
7                leafstate m4_AllocPlain [John,SysJ,Cmp,sc] = OCC []  **
7                leafstate m5_IdleWithTok [John,SysJ,Cmp,sc] = VAC []
7                leafstate m6_AllocOtherWant [John,SysJ,Cmp,sc] = VAC []
7             cluster UserJ [SysJ,Cmp,sc] = OCC []  **
7                leafstate NotHaveRes [UserJ,SysJ,Cmp,sc] = VAC []
7                leafstate Waiting [UserJ,SysJ,Cmp,sc] = VAC []
7                leafstate HaveRes [UserJ,SysJ,Cmp,sc] = OCC []  **
7          set SysM [Cmp,sc] = OCC []  **
7             cluster Mary [SysM,Cmp,sc] = OCC []  **
7                leafstate m1_IdleNoTok [Mary,SysM,Cmp,sc] = OCC []  **
7                leafstate m2_ReqdTok [Mary,SysM,Cmp,sc] = VAC []
7                leafstate m3_Waiting [Mary,SysM,Cmp,sc] = VAC []
7                leafstate m4_AllocPlain [Mary,SysM,Cmp,sc] = VAC []
7                leafstate m5_IdleWithTok [Mary,SysM,Cmp,sc] = VAC []
7                leafstate m6_AllocOtherWant [Mary,SysM,Cmp,sc] = VAC []
7             cluster UserM [SysM,Cmp,sc] = OCC []  **
7                leafstate NotHaveRes [UserM,SysM,Cmp,sc] = OCC []  **
7                leafstate Waiting [UserM,SysM,Cmp,sc] = VAC []
7                leafstate HaveRes [UserM,SysM,Cmp,sc] = VAC []
7    TRACE =[JAcq]
7    TREV [[JohnRelRes,[sc]],0,[],[ClientJohnPco,[sc]]]
7    TREV [[MaryReqTok,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
7    TREV [[beta,[sc]],0,[],[UserJPco,[sc]]]
7    TREV [[MaryReqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
7    TREV [[gamma,[sc]],0,[],[UserMPco,[sc]]]
7    TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[7]
number of outworlds=1
SC:|: pe gamma
SC:|: gc
11   statechart sc
11      set Cmp [sc] = OCC []  **
11         set SysJ [Cmp,sc] = OCC []  **
11            cluster John [SysJ,Cmp,sc] = OCC []  **
11               leafstate m1_IdleNoTok [John,SysJ,Cmp,sc] = VAC []
11               leafstate m2_ReqdTok [John,SysJ,Cmp,sc] = VAC []
11               leafstate m3_Waiting [John,SysJ,Cmp,sc] = VAC []
11               leafstate m4_AllocPlain [John,SysJ,Cmp,sc] = VAC []
11               leafstate m5_IdleWithTok [John,SysJ,Cmp,sc] = VAC []
```



```
11              leafstate m6_AllocOtherWant [John,SysJ,Cmp,sc] = OCC []  **
11           cluster UserJ [SysJ,Cmp,sc] = OCC []  **
11              leafstate NotHaveRes [UserJ,SysJ,Cmp,sc] = VAC []
11              leafstate Waiting [UserJ,SysJ,Cmp,sc] = VAC []
11              leafstate HaveRes [UserJ,SysJ,Cmp,sc] = OCC []  **
11        set SysM [Cmp,sc] = OCC []  **
11           cluster Mary [SysM,Cmp,sc] = OCC []  **
11              leafstate m1_IdleNoTok [Mary,SysM,Cmp,sc] = VAC []
11              leafstate m2_ReqdTok [Mary,SysM,Cmp,sc] = VAC []
11              leafstate m3_Waiting [Mary,SysM,Cmp,sc] = OCC []  **
11              leafstate m4_AllocPlain [Mary,SysM,Cmp,sc] = VAC []
11              leafstate m5_IdleWithTok [Mary,SysM,Cmp,sc] = VAC []
11              leafstate m6_AllocOtherWant [Mary,SysM,Cmp,sc] = VAC []
11           cluster UserM [SysM,Cmp,sc] = OCC []  **
11              leafstate NotHaveRes [UserM,SysM,Cmp,sc] = VAC []
11              leafstate Waiting [UserM,SysM,Cmp,sc] = OCC []  **
11              leafstate HaveRes [UserM,SysM,Cmp,sc] = VAC []
11   TRACE =[JAcq]
11   TREV [[JohnRelRes,[sc]],0,[],[ClientJohnPco,[sc]]]
11   TREV [[beta,[sc]],0,[],[UserJPco,[sc]]]
11   TREV [[JohnPass,[Cmp,sc]],0,[],[InterArbJohnPco,[Cmp,sc]]]
11   TREV [[MaryAcqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
11   TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[11]
number of outworlds=1
SC:|: pe beta
SC:|: gc
17   statechart sc
17      set Cmp [sc] = OCC []  **
17         set SysJ [Cmp,sc] = OCC []  **
17            cluster John [SysJ,Cmp,sc] = OCC []  **
17               leafstate m1_IdleNoTok [John,SysJ,Cmp,sc] = OCC []  **
17               leafstate m2_ReqdTok [John,SysJ,Cmp,sc] = VAC []
17               leafstate m3_Waiting [John,SysJ,Cmp,sc] = VAC []
17               leafstate m4_AllocPlain [John,SysJ,Cmp,sc] = VAC []
17               leafstate m5_IdleWithTok [John,SysJ,Cmp,sc] = VAC []
17               leafstate m6_AllocOtherWant [John,SysJ,Cmp,sc] = VAC []
17            cluster UserJ [SysJ,Cmp,sc] = OCC []  **
17               leafstate NotHaveRes [UserJ,SysJ,Cmp,sc] = OCC []  **
17               leafstate Waiting [UserJ,SysJ,Cmp,sc] = VAC []
17               leafstate HaveRes [UserJ,SysJ,Cmp,sc] = VAC []
17         set SysM [Cmp,sc] = OCC []  **
17            cluster Mary [SysM,Cmp,sc] = OCC []  **
17               leafstate m1_IdleNoTok [Mary,SysM,Cmp,sc] = VAC []
17               leafstate m2_ReqdTok [Mary,SysM,Cmp,sc] = VAC []
17               leafstate m3_Waiting [Mary,SysM,Cmp,sc] = VAC []
17               leafstate m4_AllocPlain [Mary,SysM,Cmp,sc] = OCC []  **
17               leafstate m5_IdleWithTok [Mary,SysM,Cmp,sc] = VAC []
17               leafstate m6_AllocOtherWant [Mary,SysM,Cmp,sc] = VAC []
17            cluster UserM [SysM,Cmp,sc] = OCC []  **
17               leafstate NotHaveRes [UserM,SysM,Cmp,sc] = VAC []
17               leafstate Waiting [UserM,SysM,Cmp,sc] = VAC []
17               leafstate HaveRes [UserM,SysM,Cmp,sc] = OCC []  **
17   TRACE =[MAcq,JRel,JAcq]
17   TREV [[JohnReqRes,[sc]],0,[],[ClientJohnPco,[sc]]]
17   TREV [[alpha,[sc]],0,[],[UserJPco,[sc]]]
17   TREV [[MaryRelRes,[sc]],0,[],[ClientMaryPco,[sc]]]
17   TREV [[JohnReqTok,[Cmp,sc]],0,[],[InterArbJohnPco,[Cmp,sc]]]
17   TREV [[delta,[sc]],0,[],[UserMPco,[sc]]]
17   TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[17]
number of outworlds=1
SC:|: pe delta
SC:|: gc
20   statechart sc
20      set Cmp [sc] = OCC []  **
```

```
20         set SysJ [Cmp,sc] = OCC []  **
20            cluster John [SysJ,Cmp,sc] = OCC []  **
20               leafstate m1_IdleNoTok [John,SysJ,Cmp,sc] = OCC []  **
20               leafstate m2_ReqdTok [John,SysJ,Cmp,sc] = VAC []
20               leafstate m3_Waiting [John,SysJ,Cmp,sc] = VAC []
20               leafstate m4_AllocPlain [John,SysJ,Cmp,sc] = VAC []
20               leafstate m5_IdleWithTok [John,SysJ,Cmp,sc] = VAC []
20               leafstate m6_AllocOtherWant [John,SysJ,Cmp,sc] = VAC []
20            cluster UserJ [SysJ,Cmp,sc] = OCC []  **
20               leafstate NotHaveRes [UserJ,SysJ,Cmp,sc] = OCC []  **
20               leafstate Waiting [UserJ,SysJ,Cmp,sc] = VAC []
20               leafstate HaveRes [UserJ,SysJ,Cmp,sc] = VAC []
20         set SysM [Cmp,sc] = OCC []  **
20            cluster Mary [SysM,Cmp,sc] = OCC []  **
20               leafstate m1_IdleNoTok [Mary,SysM,Cmp,sc] = VAC []
20               leafstate m2_ReqdTok [Mary,SysM,Cmp,sc] = VAC []
20               leafstate m3_Waiting [Mary,SysM,Cmp,sc] = VAC []
20               leafstate m4_AllocPlain [Mary,SysM,Cmp,sc] = VAC []
20               leafstate m5_IdleWithTok [Mary,SysM,Cmp,sc] = OCC []  **
20               leafstate m6_AllocOtherWant [Mary,SysM,Cmp,sc] = VAC []
20            cluster UserM [SysM,Cmp,sc] = OCC []  **
20               leafstate NotHaveRes [UserM,SysM,Cmp,sc] = OCC []  **
20               leafstate Waiting [UserM,SysM,Cmp,sc] = VAC []
20               leafstate HaveRes [UserM,SysM,Cmp,sc] = VAC []
20   TRACE =[MRel,MAcq,JRel,JAcq]
20   TREV [[JohnReqRes,[sc]],0,[],[ClientJohnPco,[sc]]]
20   TREV [[alpha,[sc]],0,[],[UserJPco,[sc]]]
20   TREV [[JohnReqTok,[Cmp,sc]],0,[],[InterArbJohnPco,[Cmp,sc]]]
20   TREV [[MaryReqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
20   TREV [[gamma,[sc]],0,[],[UserMPco,[sc]]]
20   TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=[20]
number of outworlds=1
SC:|:
```


3. The STATECRUNCHER distributed arbiter in CCS

The description of the model in Figure 8 is as follows, with the following renaming applied with respect to that figure:

MeReqRes → **ReqRes**
MeRelRes → **RelRes**
MeAcqRes → **AcqRes**

MeReqTok → **ReqTok**
YouReqTok → **R6e6q6T6o6k6**

MePass → **Pass**
YouPass → **P6a6s6s6**

MeSayOk → **Ok**
YouSayOk → **O6k6**

MeSayNok → **Nok**
YouSayNok → **N6o6k6**

Here is the description:

m1_IdleNoTok $=^{def}$ **ReqRes**.**ReqTok**.*m2_ReqdTok*

m2_ReqdTok $=^{def}$ **N6o6k6**.*m3_Waiting* + **P6a6s6s6**.**AcqRes**.*m4_AllocPlain*

m3_Waiting $=^{def}$ **P6a6s6s6**. **AcqRes**.*m4_AllocPlain*

m4_AllocPlain $=^{def}$ **RelRes**.*m5_IdleWithTok* + **R6e6q6T6o6k6**.**Nok**.*m6_AllocOtherWant*

m5_IdleWithTok $=^{def}$ **ReqRes**.**AcqRes**.*m4_AllocPlain*+ **R6e6q6T6o6k6**.**Pass**.*m1_IdleNoTok*

m6_AllocOtherWant $=^{def}$ **RelRes**.**Pass**.*m1_IdleNoTok*

We now consider the model given by Bruns.

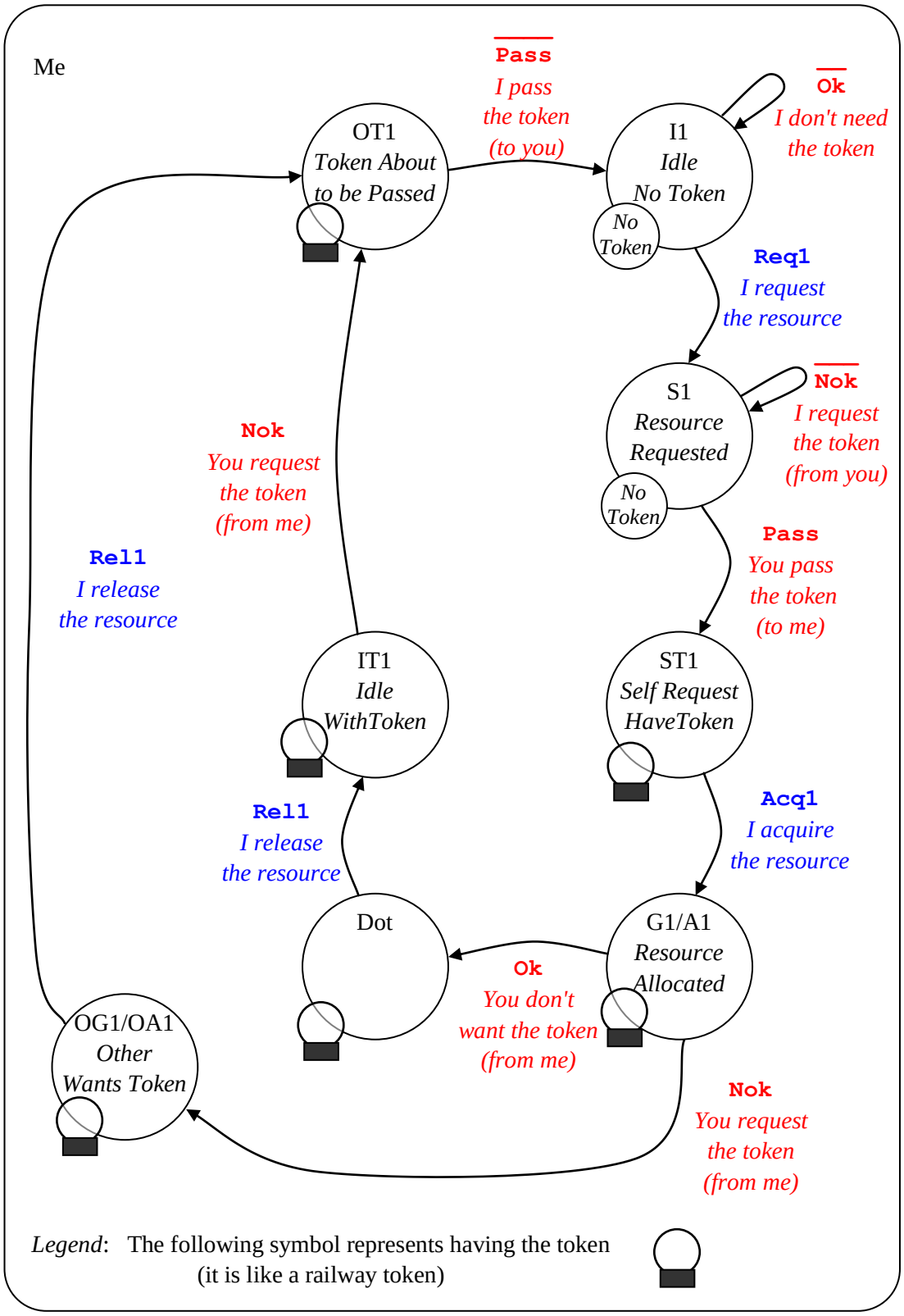


Figure 10. CCS Model in [Bruns]

Bruns's model is similar to the model in Figure 8. We make the following remarks

- Bruns has three minor typographical errors in his Figure 2.5:
  - `rel`$_1$ from state I1 should read `req`$_1$
  - state `G1` (G for Got?) should read `A1` (Allocated)
  - state `OG1` should read `OA1`
- The **`ok o4k4`** event is initiated when one arbiter acquires the resource. It is pointless, because it checks for a request for the resource when there is no need to do so. An arbiter is *told* when there is request for a resource by the **`Nok`** event; any initiative taken in *asking* about a request is also expensive (because it involves inter-arbiter communication).
- The **`Nok N6o6k6`** event is a request for the token. This will be triggered by a request for the resource when the arbiter does not have the token.

A comparison

We have named STATECRUNCHER states and events as seems natural in that language. The naming relationships between the STATECRUNCHER and Bruns's CCS model are:

- Bruns's *1* and *2* are replaced by *Me* and *You* for *events*, but we avoid such suffixes for *states*, which implicitly apply to the *Me* machine shown. When a second arbiter is introduced, states are by distinguished making names local to an arbiter.
- We distinguish **`YouPass`** and **`MePass`** as separate events
- We likewise distinguish who says *Ok* and *Nok* with **`YouSayOk`**, **`MeSayOk`**, **`YouSayNok`** *and* **`MeSayNok`**.
- Our `m2_Reqd_Tok` and `m3_Waiting` states in Figure 8 are combined by Bruns into state *S1* (resource requested).

We could merge the **`MePass`** and **`YouPass`** events into one event **`Pass`**, since no confusion would arise, but they are better considered as separate events. Similarly **the other inter-arbiter events.** They have separate origins - in separate computers even. In STATECRUNCHER it is convenient to give them separate names, since they can then be separately declared in their own machine, albeit with composition scope (through the use of scoping operators).

Bruns combines two arbiter agents with CCS calculus; STATECRUNCHER combines server and client by wrapping both in a set. STATECRUNCHER offers scoping operators for PCOs, events, states and variables so that these items can have local or composition scope (see the use of the `%%` operator in the arbiter-pair models).

Conclusion

This paper has shown how a typical client-server application is modelled in STATECRUNCHER, providing a direct comparison with a well-known example in the literature. Both STATECRUNCHER and CCS are amenable to the problem, but the emphasis is different:

STATECRUNCHER is a state machine engine providing the white box or black box the oracle to tests and does not support calculus manipulations; CCS is a *calculus* which is used to prove properties of composed systems, and is supported by Concurrency Workbench.

4. Source code of models

4.1 Source code of the single distributed arbiter [model `t4300`]

```
//-------------------------------------------------------------------------------
// Module:    d_arb.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      07 June, 2003
// Purpose:   Statecruncher model: SINGLE DISTRIBUTED ARBITER (cf Glenn Bruns CCS, p.21)
//
// Project:   Improving Component Integration
//
// Copyright (C) 2003 Philips Electronics N.V.
//
// Revision History:
//
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Me)

PCO ClientMePco;        // For client-arbiter events
PCO InterArbMePco;      // For inter-arbiter events to Me
PCO InterArbYouPco;     // For inter-arbiter events to You
// no PCO for internal events

// Acq=Acquire
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

// No need for the initial GiveMeTok event, because YouPass is legal

event MeReqRes  @ClientMePco;           // Client event to   Me
event MeRelRes  @ClientMePco;           // Client event to   Me
event MeAcqRes  @ClientMePco;           // Client event from Me

event MeReqTok  @InterArbMePco;         // InterArbiter
event MePass    @InterArbMePco;         // InterArbiter
event MeSayNok  @InterArbMePco;         // InterArbiter

event YouReqTok  @InterArbYouPco;       // InterArbiter
event YouPass    @InterArbYouPco;       // InterArbiter
event YouSayNok  @InterArbYouPco;       // InterArbiter

event TryTok;                           // Local to this arbiter
event TryOk;                            // Local to this arbiter
event AcqTok;                           // Local to this arbiter
event ResetWant;                        // Local to this arbiter

set Me(Res,Have,Need,OtherWant)

  cluster Res(Idle,Requested,Waiting,Alloc)
     state Idle      {MeReqRes->Requested {fire TryTok;}; }
     state Requested {AcqTok,TryOk->Alloc; YouSayNok->Waiting;}
```

```
    state Waiting   {AcqTok->Alloc;}
    state Alloc     {upon enter {fire MeAcqRes;} MeRelRes->Idle             \
                      {if (in($$Me.OtherWant.OtherWantTok)) {fire MePass;}};}

  cluster Have (NotHaveTok,HaveTok)
    state NotHaveTok {                          \
      TryTok {fire MeReqTok;};                  \
      YouPass->HaveTok {fire AcqTok;};          }
    state HaveTok    {                          \
      TryTok {fire TryOk;};                     \
      MePass->NotHaveTok {fire ResetWant;};     }

  cluster Need(NotNeedTok,NeedTok)
    state NotNeedTok {                          \
      YouReqTok {fire MePass;};                 \
      exit($$Me.Res.Idle)->NeedTok;             }
    state NeedTok {                             \
      YouReqTok {fire MeSayNok;};               \
      enter($$Me.Res.Idle)->NotNeedTok;         }

  cluster OtherWant(OtherNotWantTok,OtherWantTok)
    state OtherNotWantTok {                                 \
      YouReqTok [    in($$Me.Need.NeedTok)                  \
                  && in($$Me.Have.HaveTok)]->OtherWantTok;}
    state OtherWantTok    {ResetWant->OtherNotWantTok;}

//-----------------------[end of module]-----------------------------------------
```


4.2 Source code of the two distributed arbiters (John and Mary) [model t4301]

```
//-------------------------------------------------------------------------------
// Module:    d_arb.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      07 June, 2003
// Purpose:   Statecruncher model: SINGLE DISTRIBUTED ARBITER (cf Glenn Bruns CCS, p.21)
//
// Project:   Improving Component Integration
//
// Copyright (C) 2003 Philips Electronics N.V.
//
// Revision History:
//
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Me)

PCO ClientMePco;        // For client-arbiter events
PCO InterArbMePco;      // For inter-arbiter events to Me
PCO InterArbYouPco;     // For inter-arbiter events to You
// no PCO for internal events

// Acq=Acquire
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

// No need for the initial GiveMeTok event, because YouPass is legal

event MeReqRes  @ClientMePco;           // Client event to   Me
event MeRelRes  @ClientMePco;           // Client event to   Me
event MeAcqRes  @ClientMePco;           // Client event from Me

event MeReqTok  @InterArbMePco;         // InterArbiter
event MePass    @InterArbMePco;         // InterArbiter
event MeSayNok  @InterArbMePco;         // InterArbiter

event YouReqTok  @InterArbYouPco;       // InterArbiter
event YouPass    @InterArbYouPco;       // InterArbiter
event YouSayNok  @InterArbYouPco;       // InterArbiter

event TryTok;                           // Local to this arbiter
event TryOk;                            // Local to this arbiter
event AcqTok;                           // Local to this arbiter
event ResetWant;                        // Local to this arbiter

set Me(Res,Have,Need,OtherWant)

  cluster Res(Idle,Requested,Waiting,Alloc)
     state Idle      {MeReqRes->Requested {fire TryTok;}; }
     state Requested {AcqTok,TryOk->Alloc; YouSayNok->Waiting;}
     state Waiting   {AcqTok->Alloc;}
     state Alloc     {upon enter {fire MeAcqRes;} MeRelRes->Idle             \
                        {if (in($$Me.OtherWant.OtherWantTok)) {fire MePass;}};}

  cluster Have (NotHaveTok,HaveTok)
     state NotHaveTok {                         \
        TryTok {fire MeReqTok;};                \
        YouPass->HaveTok {fire AcqTok;};        }
     state HaveTok    {                         \
```

```
        TryTok {fire TryOk;};                  \
        MePass->NotHaveTok {fire ResetWant;};  }

  cluster Need(NotNeedTok,NeedTok)
     state NotNeedTok {                        \
        YouReqTok {fire MePass;};              \
        exit($$Me.Res.Idle)->NeedTok;          }
     state NeedTok {                           \
        YouReqTok {fire MeSayNok;};            \
        enter($$Me.Res.Idle)->NotNeedTok;      }

  cluster OtherWant(OtherNotWantTok,OtherWantTok)
    state OtherNotWantTok {                                  \
       YouReqTok [    in($$Me.Need.NeedTok)                  \
                   && in($$Me.Have.HaveTok)]->OtherWantTok;}
    state OtherWantTok    {ResetWant->OtherNotWantTok;}

//------------------------[end of module]-----------------------------------------------
```


4.3 Source code of the distributed arbiter with clients [model `t4302`]

```
//------------------------------------------------------------------------------
// Module:    d_arb_client.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      07 June, 2003
// Purpose:   Statecruncher model: DISTRIBUTED ARBITER WITH CLIENTS
//             (as in Glenn Bruns CCS, p.21)
//
// Project:   Improving Component Integration
//
// Copyright (C) 2003 Philips Electronics N.V.
//
// Revision History:
//
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Cmp)

PCO UserJPco;                     // For user events
PCO UserMPco;                     // For user events

PCO ClientJohnPco;                // For Client-to-arbiter events
PCO ClientMaryPco;                // For Client-to-arbiter events

// Cmp=Composition
// Acq=Acquire
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event alpha@UserJPco;                    // User event in UserJ
event beta @UserJPco;                    // User event in UserJ

event gamma@UserMPco;                    // User event in UserM
event delta@UserMPco;                    // User event in UserM

event JohnReqRes @ClientJohnPco;         // Client event   to John(arbiter)
event JohnRelRes @ClientJohnPco;         // Client event   to John(arbiter)
event JohnAcqRes @ClientJohnPco;         // Client event from John(arbiter)

event MaryReqRes @ClientMaryPco;         // Client event   to Mary(arbiter)
event MaryRelRes @ClientMaryPco;         // Client event   to Mary(arbiter)
event MaryAcqRes @ClientMaryPco;         // Client event from Mary(arbiter)

// INITIAL MANUAL EVENT TO BE GIVEN
//   It preserves symmetry between John/Mary
//      (otherwise reverse state order to get opposing default states)
//   It is GLOBAL for ease of entry
//      (we could have used event Composition%%JohnAcqTok)
event GiveJohnTok;

// ReqTok and AcqTok events are locally defined in composition scope

set Cmp(SysJ,SysM) {GiveJohnTok->Cmp.SysJ.John.Have.HaveTok;}

set SysJ(John,UserJ)

set John(Res,Have,Need,OtherWant)

    PCO   Cmp%%InterArbJohnPco;              // For inter-arbiter events to John
    event Cmp%%JohnReqTok@InterArbJohnPco;   // InterArbiter, Composition scope
    event Cmp%%JohnPass@InterArbJohnPco;     // InterArbiter, Composition scope
```

```
    event Cmp%%JohnSayNok@InterArbMaryPco;    // InterArbiter, Composition scope

    event TryTok;           // Local to this arbiter
    event TryOk;            // Local to this arbiter
    event AcqTok;           // Local to this arbiter
    event ResetWant;        // Local to this arbiter

  cluster Res(Idle,Requested,Waiting,Alloc)
     state Idle      {JohnReqRes->Requested {fire TryTok;}; }
     state Requested {AcqTok,TryOk->Alloc; MarySayNok->Waiting;}
     state Waiting   {AcqTok->Alloc;}
     state Alloc     {upon enter {fire JohnAcqRes;} JohnRelRes->Idle             \
                       {if (in($$John.OtherWant.OtherWantTok)) {fire JohnPass;}};}

  cluster Have (NotHaveTok,HaveTok)
     state NotHaveTok {                          \
        TryTok {fire JohnReqTok;};               \
        MaryPass->HaveTok {fire AcqTok;};        }
     state HaveTok    {                          \
        TryTok {fire TryOk;};                    \
        JohnPass->NotHaveTok {fire ResetWant;};  }

  cluster Need(NotNeedTok,NeedTok)
     state NotNeedTok {                          \
        MaryReqTok {fire JohnPass;};             \
        exit($$John.Res.Idle)->NeedTok;          }
     state NeedTok {                             \
        MaryReqTok {fire JohnSayNok;};           \
        enter($$John.Res.Idle)->NotNeedTok;      }

  cluster OtherWant(OtherNotWantTok,OtherWantTok)
    state OtherNotWantTok {                                  \
       MaryReqTok [   in($$John.Need.NeedTok)                \
                   && in($$John.Have.HaveTok)]->OtherWantTok;}
    state OtherWantTok    {ResetWant->OtherNotWantTok;}

cluster UserJ(NotHaveRes,Waiting,HaveRes)
   state NotHaveRes  {alpha->Waiting {fire JohnReqRes;};}
   state Waiting     {JohnAcqRes->HaveRes;}
   state HaveRes     {upon enter {trace("JAcq");}         \
                      upon exit  {trace("JRel");}         \
                      beta->NotHaveRes {fire JohnRelRes;};}

//----- as above except alpha,beta,gamma,delta and...
//                     ... Mary for John and vice versa everywhere -----

set SysM(Mary,UserM)

set Mary(Res,Have,Need,OtherWant)

    PCO   Cmp%%InterArbMaryPco;               // For inter-arbiter events to Mary
    event Cmp%%MaryReqTok @InterArbMaryPco;   // InterArbiter, Composition scope
    event Cmp%%MaryPass   @InterArbMaryPco;   // InterArbiter, Composition scope
    event Cmp%%MarySayNok @InterArbJohnPco;   // InterArbiter, Composition scope

    event TryTok;           // Local to this arbiter
    event TryOk;            // Local to this arbiter
    event AcqTok;           // Local to this arbiter
    event ResetWant;        // Local to this arbiter

  cluster Res(Idle,Requested,Waiting,Alloc)
     state Idle      {MaryReqRes->Requested {fire TryTok;}; }
     state Requested {AcqTok,TryOk->Alloc; JohnSayNok->Waiting;}
     state Waiting   {AcqTok->Alloc;}
     state Alloc     {upon enter {fire MaryAcqRes;} MaryRelRes->Idle             \
                       {if (in($$Mary.OtherWant.OtherWantTok)) {fire MaryPass;}};}

  cluster Have (NotHaveTok,HaveTok)
```



```
     state NotHaveTok {                          \
        TryTok {fire MaryReqTok;};               \
        JohnPass->HaveTok {fire AcqTok;};        }
     state HaveTok    {                          \
        TryTok {fire TryOk;};                    \
        MaryPass->NotHaveTok {fire ResetWant;};  }

  cluster Need(NotNeedTok,NeedTok)
     state NotNeedTok {                          \
        JohnReqTok {fire MaryPass;};             \
        exit($$Mary.Res.Idle)->NeedTok;          }
     state NeedTok {                             \
        JohnReqTok {fire MarySayNok;};           \
        enter($$Mary.Res.Idle)->NotNeedTok;      }

  cluster OtherWant(OtherNotWantTok,OtherWantTok)
    state OtherNotWantTok {                                  \
       JohnReqTok [   in($$Mary.Need.NeedTok)                \
                   && in($$Mary.Have.HaveTok)]->OtherWantTok;}
    state OtherWantTok    {ResetWant->OtherNotWantTok;}

cluster UserM(NotHaveRes,Waiting,HaveRes)
   state NotHaveRes  {gamma->Waiting {fire MaryReqRes;};}
   state Waiting     {MaryAcqRes->HaveRes;}
   state HaveRes     {upon enter {trace("MAcq");}          \
                      upon exit  {trace("MRel");}          \
                      delta->NotHaveRes {fire MaryRelRes;};}

//------------------------[end of module]-------------------------------------------
```

4.4 Source code of single flattened distributed arbiter [model t4310]

```
//-------------------------------------------------------------------------------
// Module:    d_arbf.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      25 June, 2003
// Purpose:   Statecruncher model: SINGLE DISTRIBUTED ARBITER FLAT MODEL
//                               (cf Glenn Bruns CCS, p.21)
//
// Project:   Improving Component Integration
//
// Copyright (C) 2003 Philips Electronics N.V.
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Me)

PCO ClientMePco;          // For client-arbiter events
PCO InterArbMePco;        // For inter-arbiter events to Me
PCO InterArbYouPco;       // For inter-arbiter events to You
// no PCO for internal events

// Acq=Acquire  (not used in flattened model)
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event MeReqRes  @ClientMePco;         // Client event to   Me
event MeRelRes  @ClientMePco;         // Client event to   Me
event MeAcqRes  @ClientMePco;         // Client event from Me

event MeReqTok @InterArbMePco;        // InterArbiter
event MePass   @InterArbMePco;        // InterArbiter
event MeSayNok @InterArbMePco;        // InterArbiter

event YouReqTok @InterArbYouPco;      // InterArbiter
event YouPass   @InterArbYouPco;      // InterArbiter
event YouSayNok @InterArbYouPco;      // InterArbiter

// No local events in the flattened model

cluster Me(  m1_IdleNoTok,   m2_ReqdTok,         \
             m3_Waiting,     m4_AllocPlain,    \
             m5_IdleWithTok, m6_AllocOtherWant )

  state m1_IdleNoTok      { MeReqRes->m2_ReqdTok           {fire MeReqTok;}; }

  state m2_ReqdTok        { YouSayNok->m3_Waiting;  \
                            YouPass->m4_AllocPlain; }

  state m3_Waiting        { YouPass->m4_AllocPlain;}

  state m4_AllocPlain     { upon enter {fire MeAcqRes;}                    \
                              MeRelRes->m5_IdleWithTok;                    \
                              YouReqTok->m6_AllocOtherWant {fire MeSayNok;}; }

  state m5_IdleWithTok    { YouReqTok->m1_IdleNoTok        {fire MePass;};   \
                            MeReqRes->m4_AllocPlain;                       }

  state m6_AllocOtherWant { MeRelRes->m1_IdleNoTok         {fire MePass;};   }


//------------------------[end of module]----------------------------------------
```


4.5 Source code of flattened distributed arbiters [model `t4311`]

```
//------------------------------------------------------------------------------
// Module:    d_arbf_pair.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      25 June, 2003
// Purpose:   Statecruncher model: TWO DISTRIBUTED ARBITERS FLATTENED
//                                 (cf Glenn Bruns CCS, p.21)
//
// Project:   Improving Component Integration
//
// Copyright (C) 2003 Philips Electronics N.V.
//
// Revision History:
//
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Cmp)

PCO ClientJohnPco;         // For user-to-arbiter events
PCO ClientMaryPco;         // For user-to-arbiter events

// Acq=Acquire  (not used in flattened model)
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event JohnReqRes @ClientJohnPco;          // Client event   to John(arbiter)
event JohnRelRes @ClientJohnPco;          // Client event   to John(arbiter)
event JohnAcqRes @ClientJohnPco;          // Client event from John(arbiter)

event MaryReqRes @ClientMaryPco;          // Client event   to Mary(arbiter)
event MaryRelRes @ClientMaryPco;          // Client event   to Mary(arbiter)
event MaryAcqRes @ClientMaryPco;          // Client event from Mary(arbiter)

// No local events in the flattened model

// INITIAL MANUAL EVENT TO BE GIVEN
event GiveJohnTok;

set Cmp(John,Mary) {                                                    \
       GiveJohnTok[in(Cmp.John.m1_IdleNoTok) && in(Cmp.Mary.m1_IdleNoTok)] \
          ->Cmp.John.m5_IdleWithTok;                                    }
       // The above condition is evaluated at execution time and does not
       // prevent the event appearing as a potential transitionable event

cluster John(  m1_IdleNoTok,   m2_ReqdTok,       \
               m3_Waiting,     m4_AllocPlain,    \
               m5_IdleWithTok, m6_AllocOtherWant )
    PCO   Cmp%%InterArbJohnPco;              // For inter-arbiter events to John
    event Cmp%%JohnReqTok @InterArbJohnPco;   // InterArbiter, Composition scope
    event Cmp%%JohnPass   @InterArbJohnPco;   // InterArbiter, Composition scope
    event Cmp%%JohnSayNok @InterArbMaryPco;   // InterArbiter, Composition scope

  state m1_IdleNoTok      { JohnReqRes->m2_ReqdTok       {fire JohnReqTok;}; }
  state m2_ReqdTok        { MarySayNok->m3_Waiting;  \
                            MaryPass->m4_AllocPlain; }
  state m3_Waiting        { MaryPass->m4_AllocPlain;}
  state m4_AllocPlain     { upon enter {fire JohnAcqRes;}                     \
                              JohnRelRes->m5_IdleWithTok;                     \
                              MaryReqTok->m6_AllocOtherWant {fire JohnSayNok;}; }
  state m5_IdleWithTok    { MaryReqTok->m1_IdleNoTok     {fire JohnPass;};   \
                            JohnReqRes->m4_AllocPlain;                      }
```

```
  state m6_AllocOtherWant { JohnRelRes->m1_IdleNoTok      {fire JohnPass;};   }

//----- as above, but Mary for John and vice versa everywhere -----

cluster Mary(  m1_IdleNoTok,   m2_ReqdTok,       \
               m3_Waiting,     m4_AllocPlain,    \
               m5_IdleWithTok, m6_AllocOtherWant )
    PCO   Cmp%%InterArbMaryPco;               // For inter-arbiter events to Mary
    event Cmp%%MaryReqTok @InterArbMaryPco;   // InterArbiter, Composition scope
    event Cmp%%MaryPass   @InterArbMaryPco;   // InterArbiter, Composition scope
    event Cmp%%MarySayNok @InterArbJohnPco;   // InterArbiter, Composition scope

  state m1_IdleNoTok      { MaryReqRes->m2_ReqdTok        {fire MaryReqTok;}; }
  state m2_ReqdTok        { JohnSayNok->m3_Waiting;  \
                            JohnPass->m4_AllocPlain; }
  state m3_Waiting        { JohnPass->m4_AllocPlain;}
  state m4_AllocPlain     { upon enter {fire MaryAcqRes;}                      \
                              MaryRelRes->m5_IdleWithTok;                      \
                              JohnReqTok->m6_AllocOtherWant {fire MarySayNok;}; }
  state m5_IdleWithTok    { JohnReqTok->m1_IdleNoTok      {fire MaryPass;};   \
                            MaryReqRes->m4_AllocPlain;                        }
  state m6_AllocOtherWant { MaryRelRes->m1_IdleNoTok      {fire MaryPass;};   }


//------------------------[end of module]-----------------------------------------
```


4.6 Source code of flattened distributed arbiter with clients [model `t4312`]

```
//------------------------------------------------------------------------------
// Module:    d_arbf_client.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      25 June, 2003
// Purpose:   Statecruncher model: DISTRIBUTED (FLAT) ARBITER WITH CLIENTS
//                                 (cf Glenn Bruns CCS, p.21)
//
// Project:   Improving Component Integration
//
// Copyright (C) 2003 Philips Electronics N.V.
//
// Revision History:
//
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Cmp)

PCO UserJPco;                    // For user events
PCO UserMPco;                    // For user events

PCO ClientJohnPco;               // For user-to-arbiter events
PCO ClientMaryPco;               // For user-to-arbiter events

// Acq=Acquire  (not used in flattened model)
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event alpha@UserJPco;                    // User event in UserJ
event beta @UserJPco;                    // User event in UserJ

event gamma@UserMPco;                    // User event in UserM
event delta@UserMPco;                    // User event in UserM

event JohnReqRes @ClientJohnPco;         // Client event   to John(arbiter)
event JohnRelRes @ClientJohnPco;         // Client event   to John(arbiter)
event JohnAcqRes @ClientJohnPco;         // Client event from John(arbiter)

event MaryReqRes @ClientMaryPco;         // Client event   to Mary(arbiter)
event MaryRelRes @ClientMaryPco;         // Client event   to Mary(arbiter)
event MaryAcqRes @ClientMaryPco;         // Client event from Mary(arbiter)

// No local events in the flattened model

// INITIAL MANUAL EVENT TO BE GIVEN
event GiveJohnTok;

set Cmp(SysJ,SysM) { GiveJohnTok                                              \
            [in(Cmp.SysJ.John.m1_IdleNoTok) && in(Cmp.SysM.Mary.m1_IdleNoTok)] \
            ->Cmp.SysJ.John.m5_IdleWithTok;                                   }
       // The above condition is evaluated at execution time and does not
       // prevent the event appearing as a potential transitionable event

set SysJ(John,UserJ)

cluster John(  m1_IdleNoTok,   m2_ReqdTok,       \
               m3_Waiting,     m4_AllocPlain,    \
               m5_IdleWithTok, m6_AllocOtherWant )
    PCO   Cmp%%InterArbJohnPco;              // For inter-arbiter events to John
```

```
    event Cmp%%JohnReqTok @InterArbJohnPco;   // InterArbiter, Composition scope
    event Cmp%%JohnPass   @InterArbJohnPco;   // InterArbiter, Composition scope
    event Cmp%%JohnSayNok @InterArbMaryPco;   // InterArbiter, Composition scope

  state m1_IdleNoTok      { JohnReqRes->m2_ReqdTok        {fire JohnReqTok;}; }
  state m2_ReqdTok        { MarySayNok->m3_Waiting;  \
                            MaryPass->m4_AllocPlain; }
  state m3_Waiting        { MaryPass->m4_AllocPlain;}
  state m4_AllocPlain     { upon enter {fire JohnAcqRes;}                      \
                              JohnRelRes->m5_IdleWithTok;                      \
                              MaryReqTok->m6_AllocOtherWant {fire JohnSayNok;}; }
  state m5_IdleWithTok    { MaryReqTok->m1_IdleNoTok      {fire JohnPass;};   \
                            JohnReqRes->m4_AllocPlain;                       }
  state m6_AllocOtherWant { JohnRelRes->m1_IdleNoTok      {fire JohnPass;};   }

cluster UserJ(NotHaveRes,Waiting,HaveRes)
   state NotHaveRes  {alpha->Waiting {fire JohnReqRes;};}
   state Waiting     {JohnAcqRes->HaveRes;}
   state HaveRes     {upon enter {trace("JAcq");}         \
                      upon exit  {trace("JRel");}         \
                      beta->NotHaveRes {fire JohnRelRes;};}

//----- as above, but Mary for John and vice versa everywhere -----

set SysM(Mary,UserM)

cluster Mary(  m1_IdleNoTok,   m2_ReqdTok,       \
               m3_Waiting,     m4_AllocPlain,    \
               m5_IdleWithTok, m6_AllocOtherWant )
    PCO   Cmp%%InterArbMaryPco;               // For inter-arbiter events to Mary
    event Cmp%%MaryReqTok @InterArbMaryPco;   // InterArbiter, Composition scope
    event Cmp%%MaryPass   @InterArbMaryPco;   // InterArbiter, Composition scope
    event Cmp%%MarySayNok @InterArbJohnPco;   // InterArbiter, Composition scope

  state m1_IdleNoTok      { MaryReqRes->m2_ReqdTok        {fire MaryReqTok;}; }
  state m2_ReqdTok        { JohnSayNok->m3_Waiting;  \
                            JohnPass->m4_AllocPlain; }
  state m3_Waiting        { JohnPass->m4_AllocPlain;}
  state m4_AllocPlain     { upon enter {fire MaryAcqRes;}                      \
                              MaryRelRes->m5_IdleWithTok;                      \
                              JohnReqTok->m6_AllocOtherWant {fire MarySayNok;}; }
  state m5_IdleWithTok    { JohnReqTok->m1_IdleNoTok      {fire MaryPass;};   \
                            MaryReqRes->m4_AllocPlain;                       }
  state m6_AllocOtherWant { MaryRelRes->m1_IdleNoTok      {fire MaryPass;};   }

cluster UserM(NotHaveRes,Waiting,HaveRes)
   state NotHaveRes  {gamma->Waiting {fire MaryReqRes;};}
   state Waiting     {MaryAcqRes->HaveRes;}
   state HaveRes     {upon enter {trace("MAcq");}          \
                      upon exit  {trace("MRel");}          \
                      delta->NotHaveRes {fire MaryRelRes;};}

//------------------------[end of module]-----------------------------------------
```


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

[CWB] The Edinburgh Concurrency Workbench
http://www.dcs.ed.ac.uk/home/cwb/

[Harel 87] D. Harel *et al.*
On the Formal Semantics of Statecharts
Logic in Computer Science, 2nd Annual Conference, 1987, pp.54-64

[Milner] Robin Milner
Communication and Concurrency
Prentice Hall 1997, ISBN 0-13-114984-9 and 0-13-115007-3 Pbk

[UML] The Object Management Group website is: http://www.omg.org
UML specifications are available from this website.

