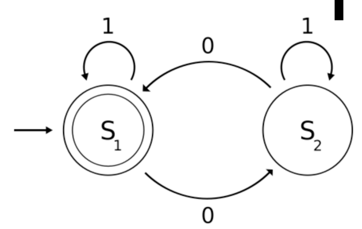
Becoming `textual`: attempting to model ‘XCHAN’ with `CSPm`



Using FDR2 and ProBE tools when state-ing is not enough

Øyvind Teig, Autronica Fire and Security

http://www.teigfam.net/oyvind/home/

Lecture material at:
http://www.teigfam.net/oyvind/home/technology/063-lecture-ntnu/

Refinement | Deadlock | Livelock | Determinism | Evaluate

```
P_XCHAN = (
    xchan_ready_ ! ready_sender_has_xmessage ->
    xchan_leg2_  ! commit_discard_xmessage.xmessage ->
    xchan_ready_ ! ready_send_now ->
    (
       xchan_leg1_ ? piped_through.xmessage ->
           xchan_leg2_ ! piped_through.xmessage ->
           P_XCHAN
       []
       xchan_leg1_ ? newest_after_overflow.xmessage ->
           xchan_leg2_ ! newest_after_overflow.xmessage ->
           P_XCHAN
    )
)
```

Exam lecture for

TTK3 - Sanntidsteori, Real-time theory (1)

in the spirit of

TK8112 - The Theory of Concurrency in Real-Time Systems (2)

Trondheim, 15. April 2013 (Electrical Engineering D240 12:15-14:00) ->
(Rev2, after exam same date: typos fixed and new layout on References page)
(Rev3, August 2013: the two pages of «Modeling XCHAN" have been updated)

Introduction

1. **Introduction**
2. Theory: XCHAN
3. Hands on: deadlock
4. Determinism-analysis of the XCHAN model
5. Conclusion

Meeting the requirements

- What *is* a requirement and what *is* an implementation?
- How do we know that an implementation fulfills a requirement?

CSPm (also called CSP_M)

- CSPm (3),(4) is a scripting language that combines CSP process algebra with an expression language to support the idioms of CSP

- The three operators **? ! .** bind names to values in the functional language part of CSPm. There are no explicit assignments, but there are «Datatype» definitions

- **? !** are *syntactic sugar*. «There is no sending, no receiving - just synchronizing on an event and optional exchange of data.
  c?x -> P(x) is syntactic sugar for "will synchronise on any event
  c.a ∊ {|c|}, binding the name x to each a in the subsequent process definition"» (in letter from P. A., UofOx)

- Algorithms may be modeled in CSP, not «executed», only shown that they may be executed (the terms«executable» as used in Promela is not used)

- Not everything in the book (12) (Roscoe) is implemented in CSPm - f.ex. «synchronous parallel». Same terms may even have different names. See my blog note (5) for a discussion

FDR2

- Compiles CSPm scripts. Is Formal System’s «heavy» tool
- I installed it on OSX (Mac OS X) binaries. Again, see my blog note (5)
- Uses X11 (XQuartz on OSX)
- Presently beta testing a new version at University of Oxford (source: UofOx)

ProBE

- Also compiles CSPm scripts
- Is «an animator for CSP processes allowing the user to explore the behaviour of models interactively»
- I discovered that the download link was dead, and when Oxford had been made aware of this the binaries were restored on 1March2013
- I downloaded the vintage Win95 version, as there was no OSX version. Runs under WineApp.app on OSX, as does the folding editor WinF. Again, see the blog note (5).
- Proved to be as promising as I had hoped for during my 1-2 moths of FDR-only despair. Opened up for a lot of aha-experiences

Self study

1. After this lecture, you should be able to

2. Install and run FDR2 and ProBE

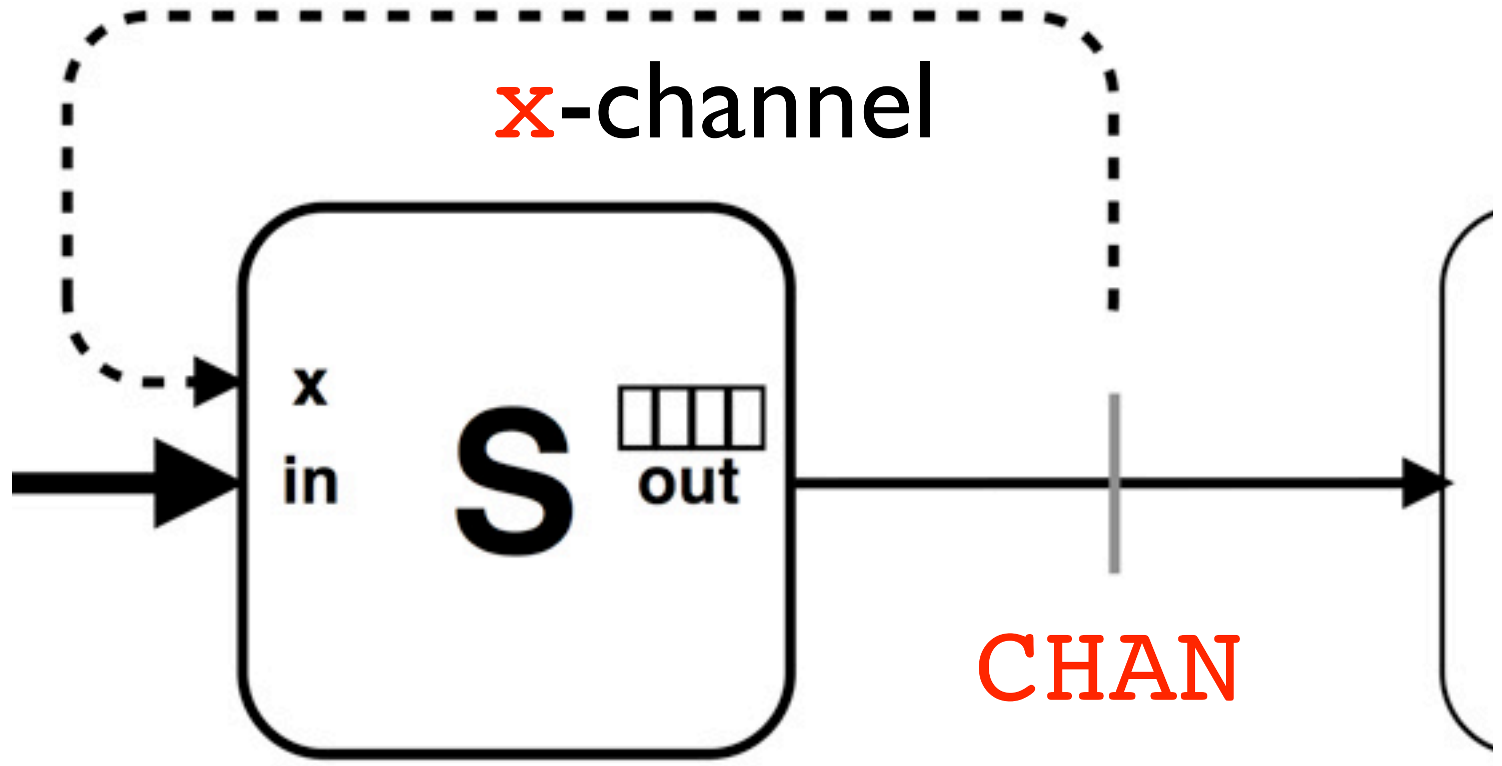
3. Do self study of `mbuff.csp` which is covered as a tutorial in the FDR2 User Manual (6). See «1.4 Specification Example», «1.4.1 Multiplexed buffer example» and «3.2.2 Getting started». I started with this, but will not go throught it here

4. Continue with other files in the 'demo' directory. I assume they have been carefully selected to take the student through most of the secret paths. Many of these have also been described in the lecture book (12) (Roscoe)

Theory: XCHAN

XCHAN [1]

x-channel + CHAN



XCHANs: Notes on a New Channel Type, in Communicating Process Architectures 2012. See (8)

Why XCHAN here?

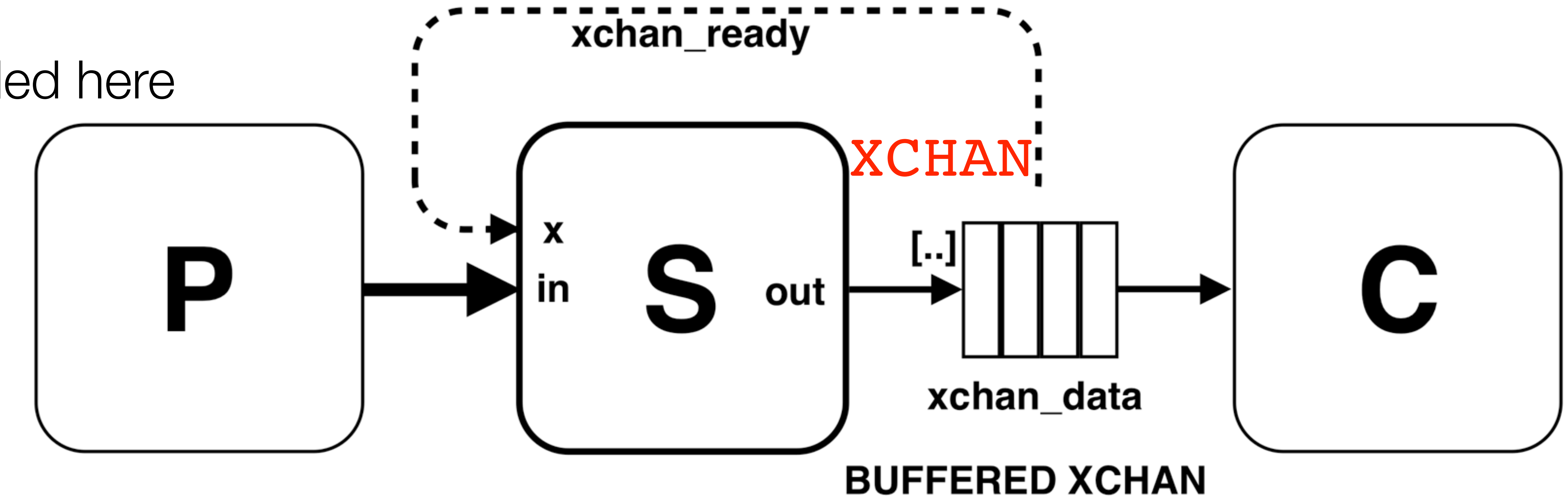
- XCHAN by itself is not relevant to this lecture
- However, going from an English word description (specification) and trying to model it in CSPm and verifying the model with FDR2 and ProBE is relevant to this lecture
- XCHAN was «my case» that easily motivated me
- After having learned from my struggling here, try to find your own case

Why XCHAN here?

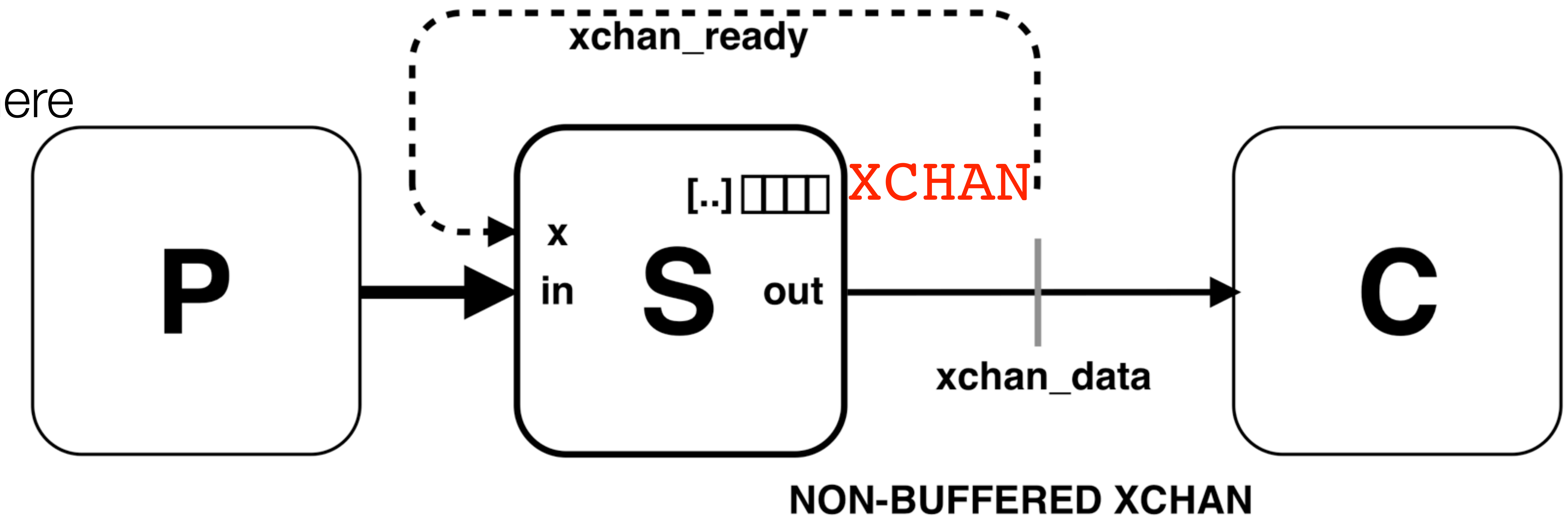
- XCHAN by itself is not relevant to this lecture
- However, going from an English word description (specification) and trying to model it in CSPm and verifying the model with FDR2 and ProBE is relevant to this lecture
- XCHAN was «my case» that easily motivated me
- After having learned from my struggling here, try to find your own case
- ..or try to model XCHAN *simpler* and *better* (then mail me)

XCHAN [2]

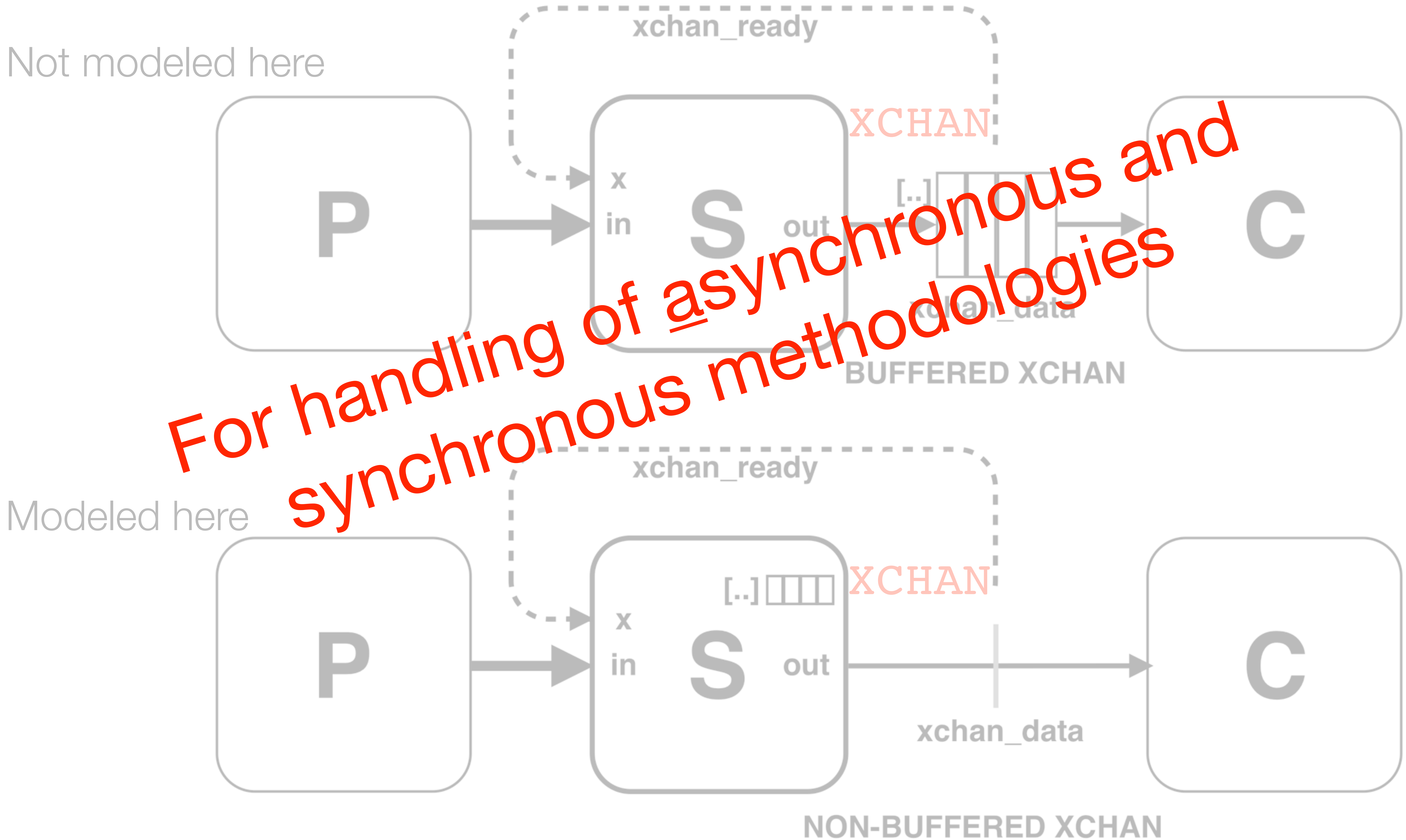
Not modeled here



Modeled here



XCHAN [2]


Not modeled here
xchan_ready
XCHAN
P
x
in
S
out
[..]
C
xchan_data
BUFFERED XCHAN
Modeled here
xchan_ready
XCHAN
[..]
P
x
in
S
out
C
xchan_data
NON-BUFFERED XCHAN
For handling of asynchronous and synchronous methodologies

Modeling XCHAN

Prof. Peter Welch made several models of buffered and unbuffered XCHAN in occam-pi during proof-reading of the original XCHAN paper (*). I have photos of the listings he showed me at CPA-2012 (**), but here is a summary:

1. An occam process model of a buffered XCHAN, including a modified standard ring buffer (xchan.occ)

2. An occam process model of an unbuffered XCHAN. Two versions:

    a. Uses non-implemented **!!**, **!!** extended output and input **??**, **??** (tho phase write)

    b. Uses two explicit readings on XCHAN end (first to exit ALT, second to pick data)

(*) In my paper I had done reasoning to show that XCHAN is implementable

(**) The model was presented at the *fringe* at CPA-2103 (the year after)
An occam Model of XCHANs
Peter H. WELCH (a) and Øyvind TEIG (b)
(a) School of Computing, University of Kent, UK
(b) Autronica Fire and Security AS, Trondheim, Norway
See http://wotug.org/cpa2013/programme.shtml#paper63

ASIDE: xchan-ready-first or xchan-ready-classic

- All of Peter Welch's senders get xchan-ready (true) when the connection with the receiver was committed. After xchan-ready (true) the sender must send, and this is the only place to send. This algorithm also fully implements the original XCHAN semantics. We could call this the «**preconfirmed**» solution

- The original XCHAN paper may start sending any time, but if sending fails then the xchan-ready is signalled when the connection with the receiver is fully committed. So, this «**classic**» solution only uses xchan-ready to send after an initial failure (*)

(*) At CPA-2013 I published a paper about «feathering», which in fact needs «classic» XCHAN semantics:
Selective choice 'feathering' with XCHANs
Communicating Process Architectures 2013 (CPA-2013)
See http://www.teigfam.net/oyvind/pub/pub_details.html#FEATHERING

Repeated CSPm back to square one

- I tried to model XCHAN in CSPm as best as I could but for a long time I failed to understand the landscape:
  - ..because I tried to look for Lego bricks that don’t exist
- I continuously had to go back to square one
- Being new to this I even tried to write a «test program» instead of a specification
  - A test program that sends data and analyses the output to see if they are correct is not a specification!
  - A specification describes what the implemenation must do in a more general way
It is not a test program!

Repeated CSPm back to square one

- I tried to model XCHAN in CSPm as best as I could but for a long time I failed to understand the landscape:
  - ..because I tried to look for Lego bricks that don't exist
- I continuously had to go back to square one
- Being new to this I even tried to write a «test program» instead of a specification
  - A test program that sends data and analyses the output to see if they are correct is not a specification!
  - A specification describes what the implemenation must do in a more general way
    It is not a test program!

But can't I specify what I need?

Repeated CSPm back to square one

- I tried to model XCHAN in CSPm as best as I could but for a long time I failed to understand the landscape:
    - ..because I tried to look for Lego bricks that don't exist
- I continuously had to go back to square one
- Being new to this I even tried to write a «test program» instead of a specification
    - A test program that sends data and analyses the outputs to see if they are correct is not a specification!
    - A specification describes what the implemenation must do in a more general way
It is not a test program!

Like specifying a test program and the system under test?

But maybe a specification could be of a system under test?

Problems(?)

1. Writing a specification that as a consequence of a fast producer and slow consumer will sooner or later lose data

    - CSPm has no concept of time, nor any delay. I cannot say something like «during a burst chan_left must accept one input every tick, but chan_right only accepts one output on every 5th tick». If so, a buffer of 5 would store for 5-6 ticks without overflow. I don't know which buttons to press in CSPm to specify anything like this. And, is there another way to say the same?

    - Still I have not resolved what *delayed choice* and *untimed timeout* can do for me. They are really undocumented

    - Timed CSP (9) or PAT (11) could perhaps be used for needs like this?

2. Writing a specification that would normally <u>pipe all data</u> through, but may alternatively <u>lose all data</u>

    - CSPm has no prioritised choice that would make it possible for me to check chan_ready «first», if there was nothing there, then chan_left would be included in the choice

3. But will my final result here show that for an XCHAN system I won't need any of the above?

Solutions(?)

I dreamt up more and more difficult solutions. Like

- trying to simulate prioritised choice (by feedback?)
- I though I had simulated this in one end of the model, but then, on the other end I failed
- It became unmanagable for me. That’s when square one was good to have

Repeated CSPm but *not* back to 1?

- Realising what CSPm offers and does not offer is in the learning
- Only recently in this process ProBE appeared, and it made me see and then understand more
- Learing to reason about a subpart of the system and see that it is enough that this part is asserted true in a verification, is enough!
- Starting to discover the Lego bricks and their roles: refinement, failures, failure-divergence, traces, deadlock, livelock and determinism. Hiding (and renaming)
- Starting to see the basics of CSPm slowly takes me by the hand and leads me to a next level

The model(s) architecture

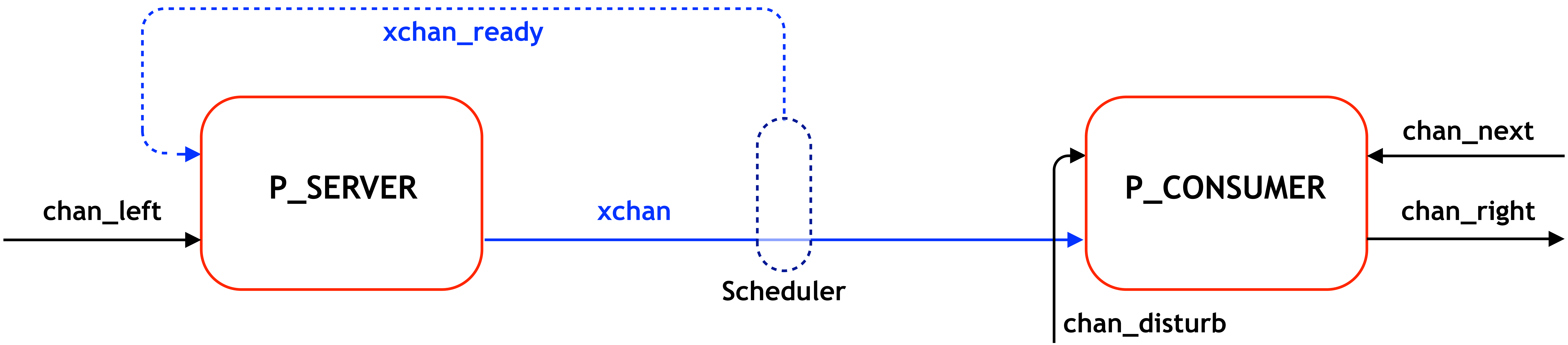


Figure 1

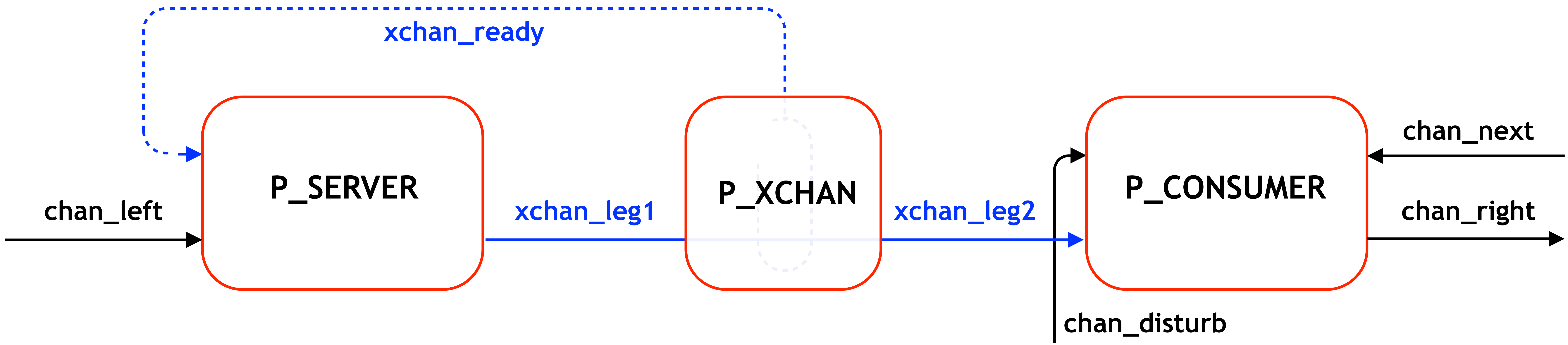

xchan_ready
P_SERVER
chan_left
xchan_leg1
P_XCHAN
xchan_leg2
P_CONSUMER
chan_next
chan_right
chan_disturb

Figure 2

```
datatype Prot_Message      = xmessage
datatype Prot_Message_Tag = piped_through | newest_after_overflow | commit_discard_xmessage
datatype Prot_Next_Output = next_out
datatype Prot_Ready        = ready_sender_has_xmessage | ready_send_now
datatype Prot_Disturb      = disturb
datatype Num_Received      = none | one | several

channel chan_main_in_   : Prot_Message
channel chan_main_out_  : Prot_Message_Tag.Prot_Message
channel chan_left_      : Prot_Message
channel chan_mid_       : Prot_Message
channel chan_right_     : Prot_Message_Tag.Prot_Message
channel xchan_leg1_     : Prot_Message_Tag.Prot_Message
channel xchan_leg2_     : Prot_Message_Tag.Prot_Message
channel xchan_ready_    : Prot_Ready
channel chan_next_      : Prot_Next_Output
channel chan_disturb_   : Prot_Disturb
```

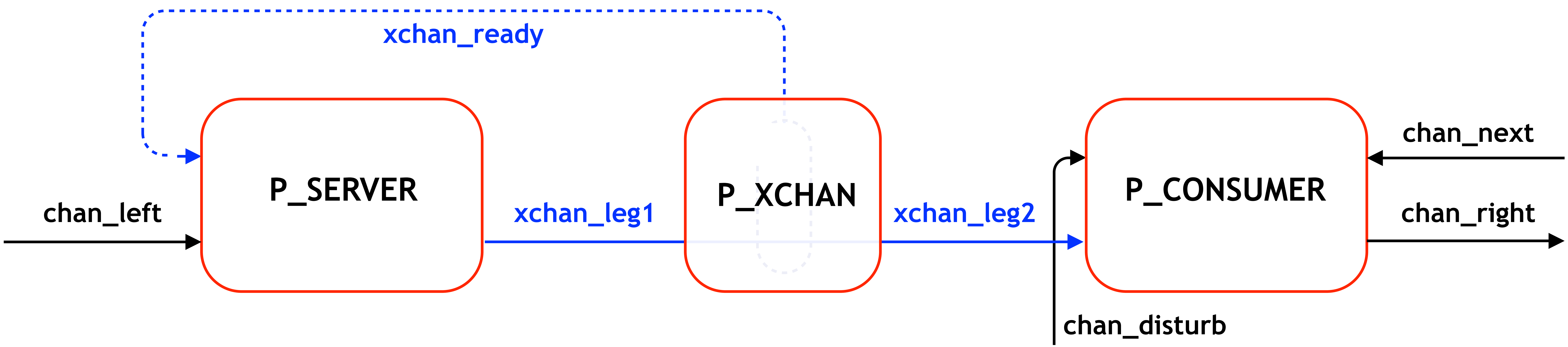


Figure 2

SYNCHRONISES THE SENDER AND RECEIVER END OF AN XCHANNEL BY EXPOSING THE INNER STATE CHANGES TO THE PARTIES

```
P_XCHAN = (
    xchan_ready_ ! ready_sender_has_xmessage ->
    xchan_leg2_  ! commit_discard_xmessage.xmessage ->
    xchan_ready_ ! ready_send_now ->
    (
       xchan_leg1_ ? piped_through.xmessage ->
           xchan_leg2_ ! piped_through.xmessage ->
           P_XCHAN
       []
       xchan_leg1_ ? newest_after_overflow.xmessage ->
           xchan_leg2_ ! newest_after_overflow.xmessage ->
           P_XCHAN
    )
)
```

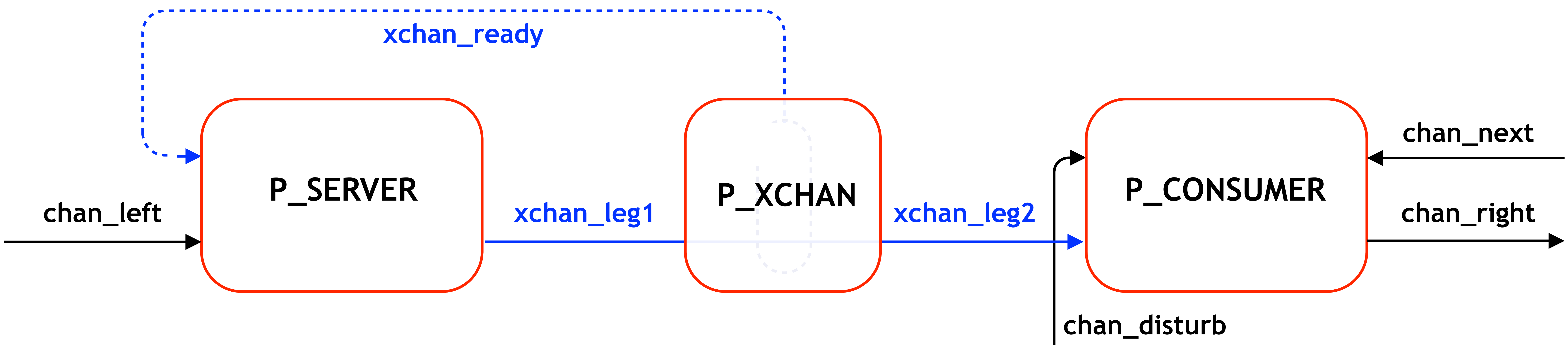


Figure 2

SYNCHRONISES THE SENDER AND RECEIVER END OF AN XCHANNEL BY EXPOSING THE INNER STATE CHANGES TO THE PARTIES

expected value

actual value

```
P_XCHAN = (
    xchan_ready_ ! ready_sender_has_xmessage ->
    xchan_leg2_  ! commit_discard_xmessage.xmessage ->
    xchan_ready_ ! ready_send_now ->
    (
       xchan_leg1_ ? piped_through.xmessage ->
          xchan_leg2_ ! piped_through.xmessage ->
          P_XCHAN
       []
       xchan_leg1_ ? newest_after_overflow.xmessage ->
          xchan_leg2_ ! newest_after_overflow.xmessage ->
          P_XCHAN
    )
)
```

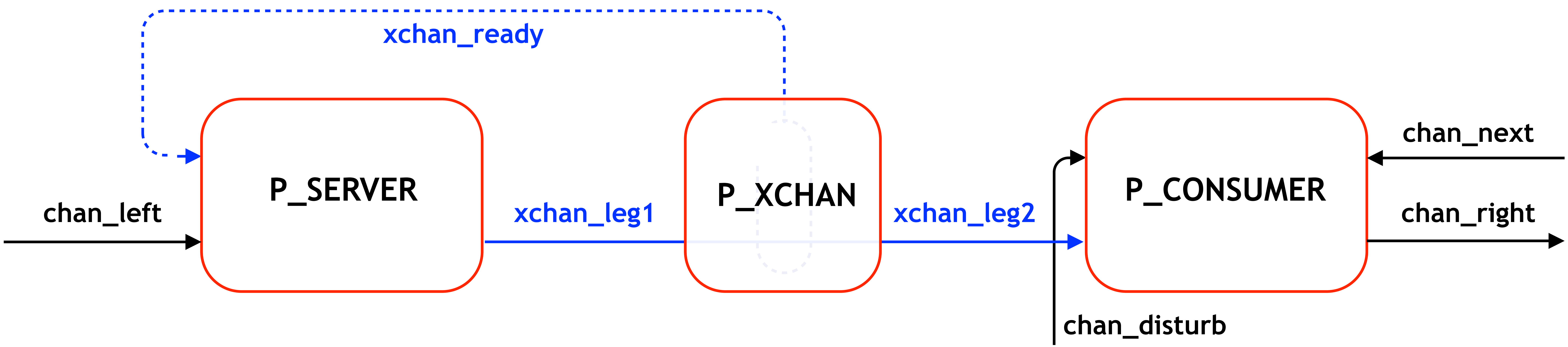


Figure 2

```
P_SERVER = (
    let
        P_SERVER_5 (this_num_received, this_wait_for_receiver, this_do_output, this_xmessage_tag, this_xmessage) = (

            ((this_num_received != none) and (not this_wait_for_receiver) and (not this_do_output)) &
            xchan_ready_ ? ready_sender_has_xmessage -> (
                P_SERVER_5 (this_num_received, true, this_do_output, this_xmessage_tag, this_xmessage)
            )
            []
            this_wait_for_receiver & xchan_ready_ ? ready_send_now ->
                P_SERVER_5 (this_num_received, false, true, this_xmessage_tag, this_xmessage)
            []
            this_do_output & xchan_leg1_ ! this_xmessage_tag.this_xmessage ->
                P_SERVER_5 (none, false, false, null, xmessage)
            []
            chan_left_ ? xmessage -> (
                if (this_num_received == none) then (
                    P_SERVER_5 (one, this_wait_for_receiver, this_do_output, piped_through, xmessage)
                ) else (
                    P_SERVER_5 (several, this_wait_for_receiver, this_do_output, newest_after_overflow, xmessage)
                )
            )
        )
    within P_SERVER_5 (none, false, false, null, xmessage)
)
```

ALWAYS INPUTS MESSAGES AND TRIES TO OUTPUT THEM ON THE XCHANNEL AND HANDLES OVERFLOW AT P_SERVER APPLICATION LEVEL

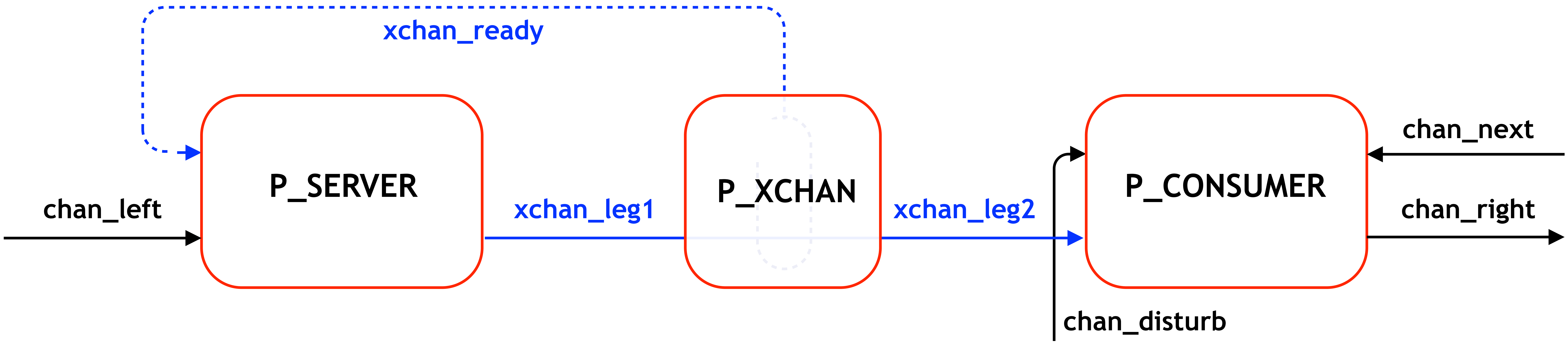


Figure 2

```
P_SERVER = (
    let
        P_SERVER_5 (this_num_received, this_wait_for_receiver, this_do_output, this_xmessage_tag, this_xmessage) = (

            ((this_num_received != none) and (not this_wait_for_receiver) and (not this_do_output)) &
            xchan_ready_ ? ready_sender_has_xmessage -> (
                P_SERVER_5 (this_num_received, true, this_do_output, this_xmessage_tag, this_xmessage)
            )
            []
            this_wait_for_receiver & xchan_ready_ ? ready_send_now ->
                P_SERVER_5 (this_num_received, false, true, this_xmessage_tag, this_xmessage)
            []
            this_do_output & xchan_leg1_ ! this_xmessage_tag.this_xmessage ->
                P_SERVER_5 (none, false, false, null, xmessage)
            []
            chan_left_ ? xmessage -> (
                if (this_num_received == none) then (
                    P_SERVER_5 (one, this_wait_for_receiver, this_do_output, piped_through, xmessage)
                ) else (
                    P_SERVER_5 (several, this_wait_for_receiver, this_do_output, newest_after_overflow, xmessage)
                )
            )
        )
    within P_SERVER_5 (none, false, false, null, xmessage)
)
```

ALWAYS INPUTS MESSAGES AND TRIES TO OUTPUT THEM ON THE XCHANNEL AND HANDLES OVERFLOW AT P_SERVER APPLICATION LEVEL

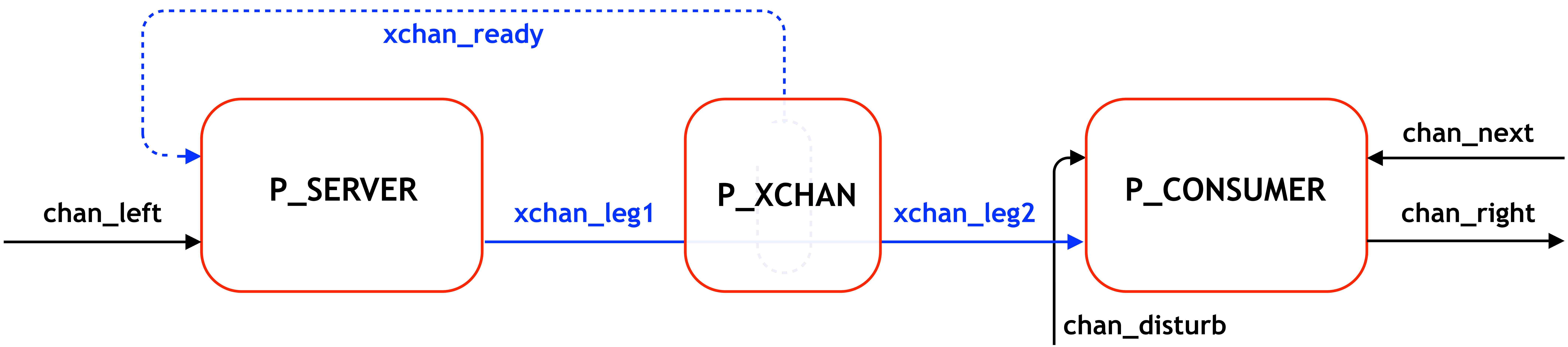


Figure 2

TAKES INPUT ON THE XCHANNEL WHEN IT IS ALLOWED BY HANDSHAKE TO GET RID OF IT

```
P_CONSUMER = (
    let
        P_CONSUMER_3 (this_num_received, this_next_sequence, this_xmessage) = (
           (this_next_sequence == 0) & chan_next_ ? next_out ->
               P_CONSUMER_3 (this_num_received, 1, this_xmessage)
           []
           (this_next_sequence == 1) & xchan_leg2_ ? commit_discard_xmessage.xmessage ->
               P_CONSUMER_3 (this_num_received, 2, xmessage)
           []
           (this_next_sequence == 2) & xchan_leg2_ ? piped_through.xmessage ->
               P_CONSUMER_3 (one, 3, xmessage)
           []
           (this_next_sequence == 2) & xchan_leg2_ ? newest_after_overflow.xmessage ->
               P_CONSUMER_3 (several, 3, xmessage)
           []
           ((this_next_sequence == 3) and (this_num_received == one)) & chan_right_ ! piped_through.this_xmessage ->
               P_CONSUMER_3 (none, 0, null)
           []
           ((this_next_sequence == 3) and (this_num_received == several)) & chan_right_ ! newest_after_overflow.this_xmessage ->
               P_CONSUMER_3 (none, 0, null)
           []
           chan_disturb_ ? disturb ->
               P_CONSUMER_3 (this_num_received, this_next_sequence, this_xmessage)
        )
    within P_CONSUMER_3 (none, 0, null)
)
```

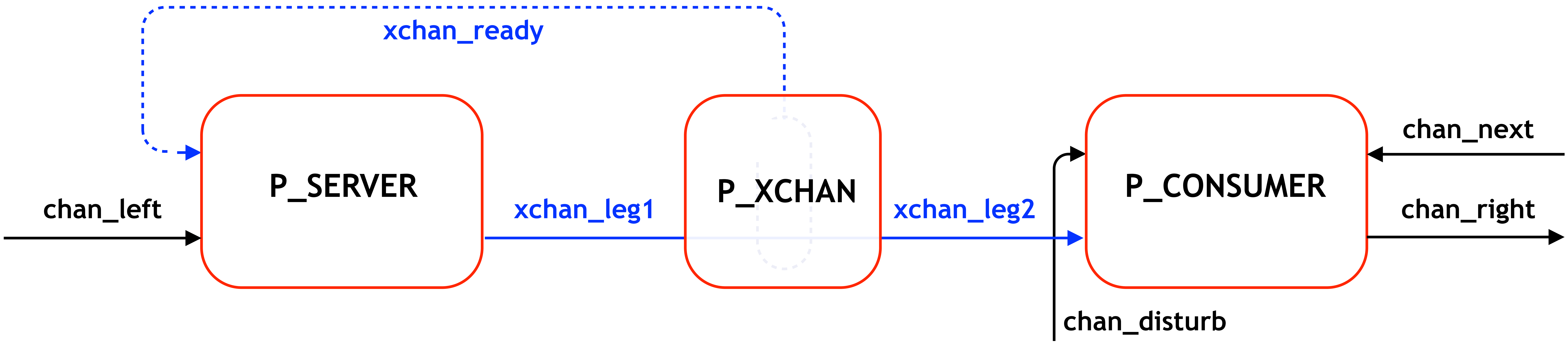


Figure 2

EXERCISES THE XCHANNEL AND ALSO CONTAINS THE P_XCHAN HANDLING PROCESS

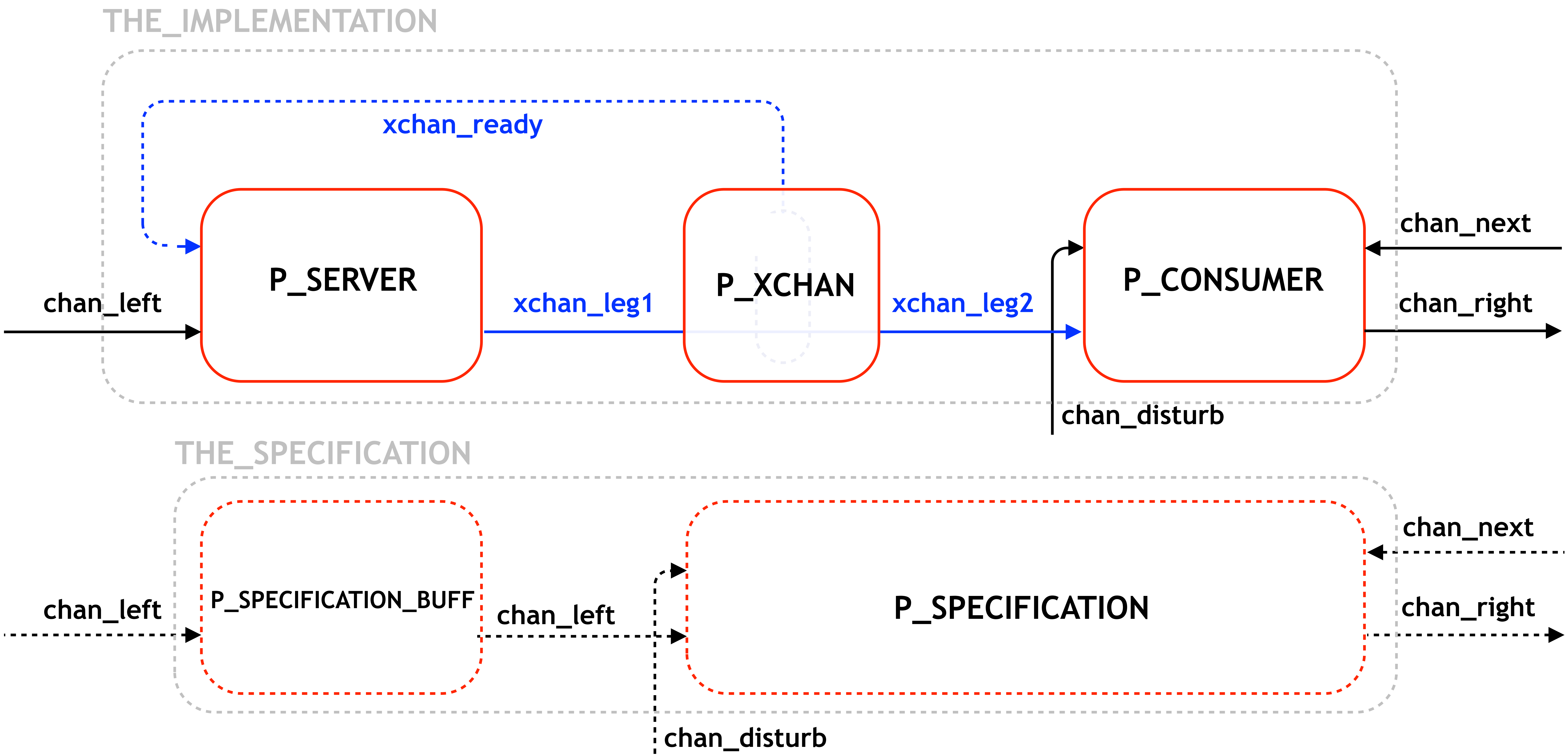


Figure 3

IS A COMPOSITE PROCESS OF THE_IMPLEMENTATION AND P_TESTER

THE_IMPLEMENTATION_OUTER

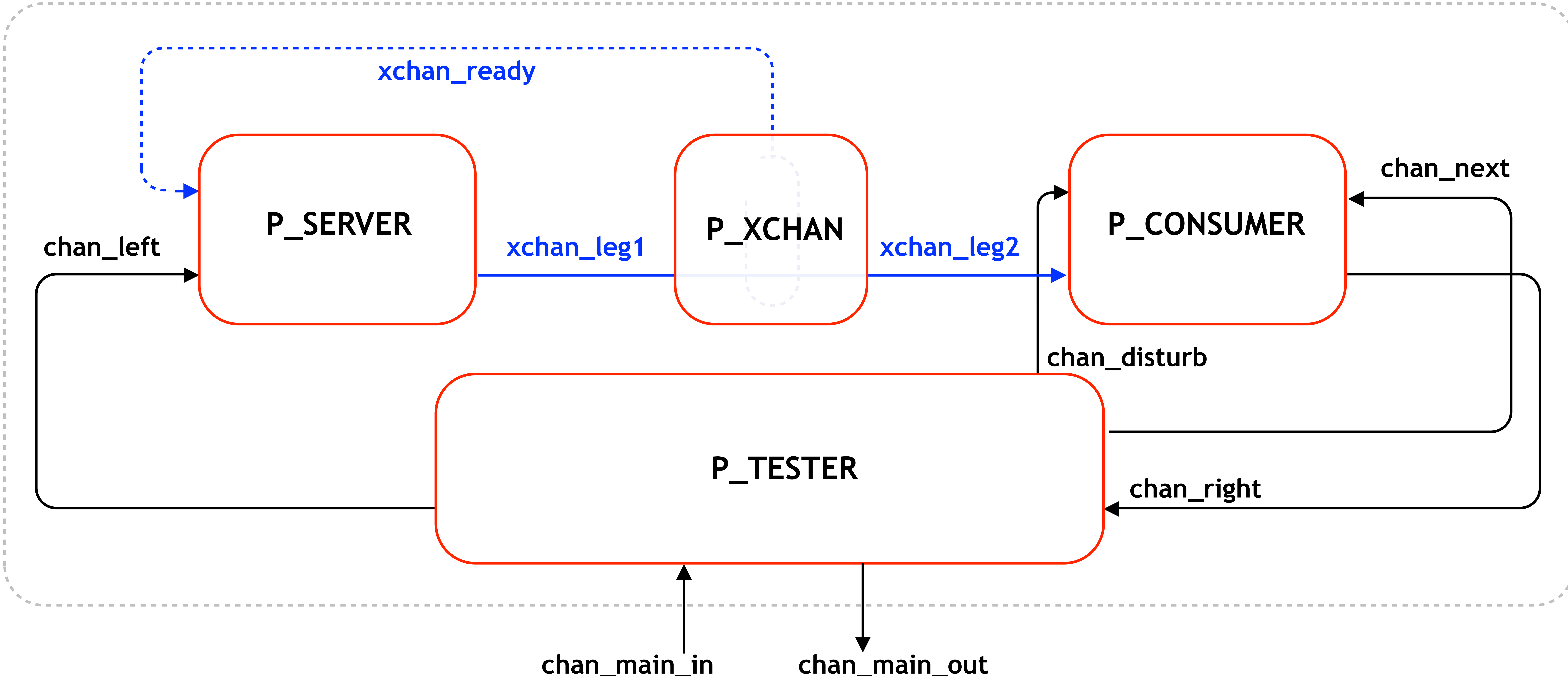


Figure 4

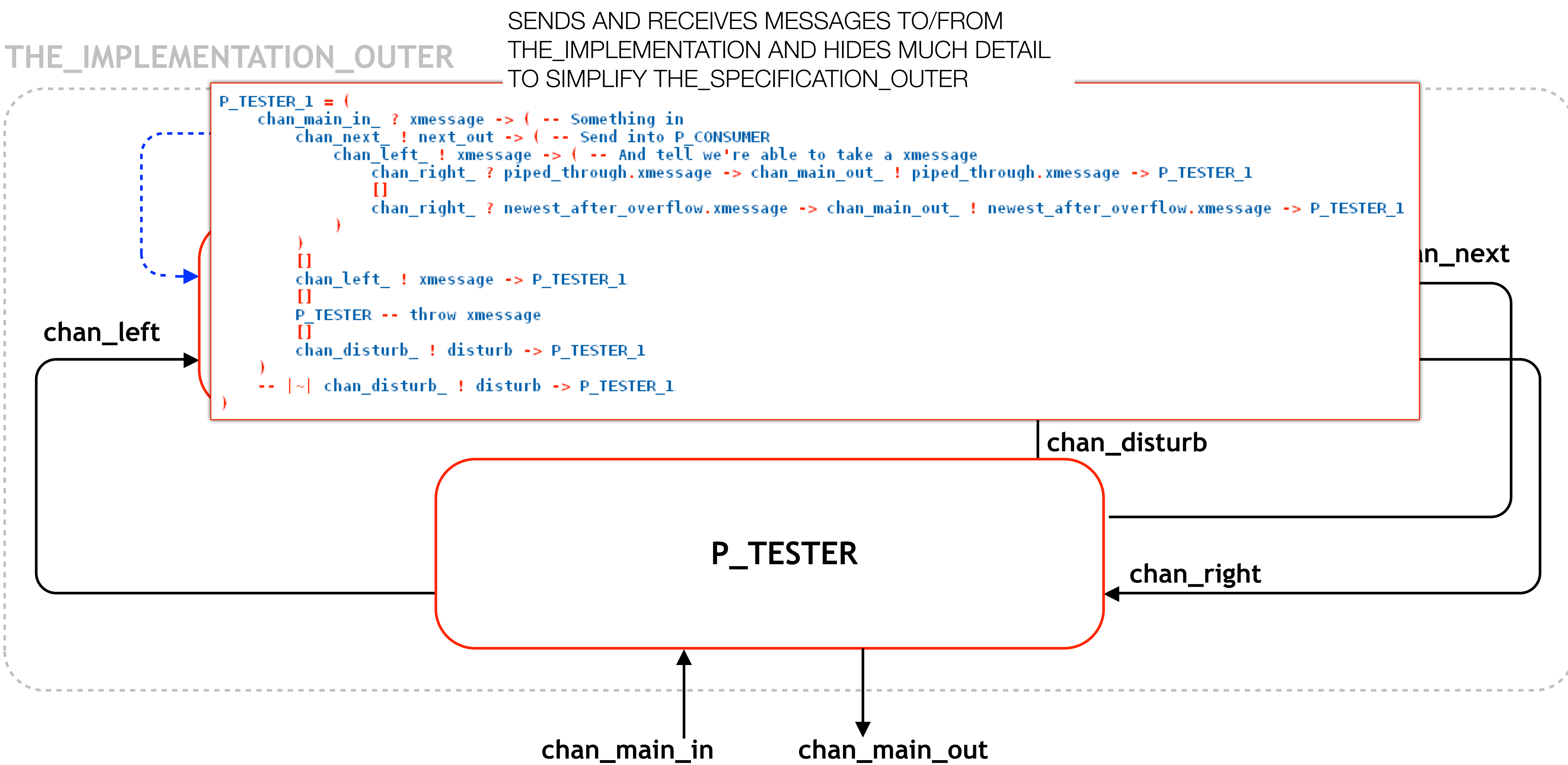

SENDS AND RECEIVES MESSAGES TO/FROM
THE_IMPLEMENTATION AND HIDES MUCH DETAIL
TO SIMPLIFY THE_SPECIFICATION_OUTER
THE_IMPLEMENTATION_OUTER
P_TESTER_1 = (
chan_main_in_ ? xmessage -> (-- Something in
chan_next_ ! next_out -> (-- Send into P_CONSUMER
chan_left_ ! xmessage -> (-- And tell we're able to take a xmessage
chan_right_ ? piped_through.xmessage -> chan_main_out_ ! piped_through.xmessage -> P_TESTER_1
[]
chan_right_ ? newest_after_overflow.xmessage -> chan_main_out_ ! newest_after_overflow.xmessage -> P_TESTER_1
)
)
[]
chan_left_ ! xmessage -> P_TESTER_1
[]
P_TESTER -- throw xmessage
[]
chan_disturb_ ! disturb -> P_TESTER_1
)
-- |~| chan_disturb_ ! disturb -> P_TESTER_1
)
chan_left
n_next
chan_disturb
P_TESTER
chan_right
chan_main_in
chan_main_out

Figure 4

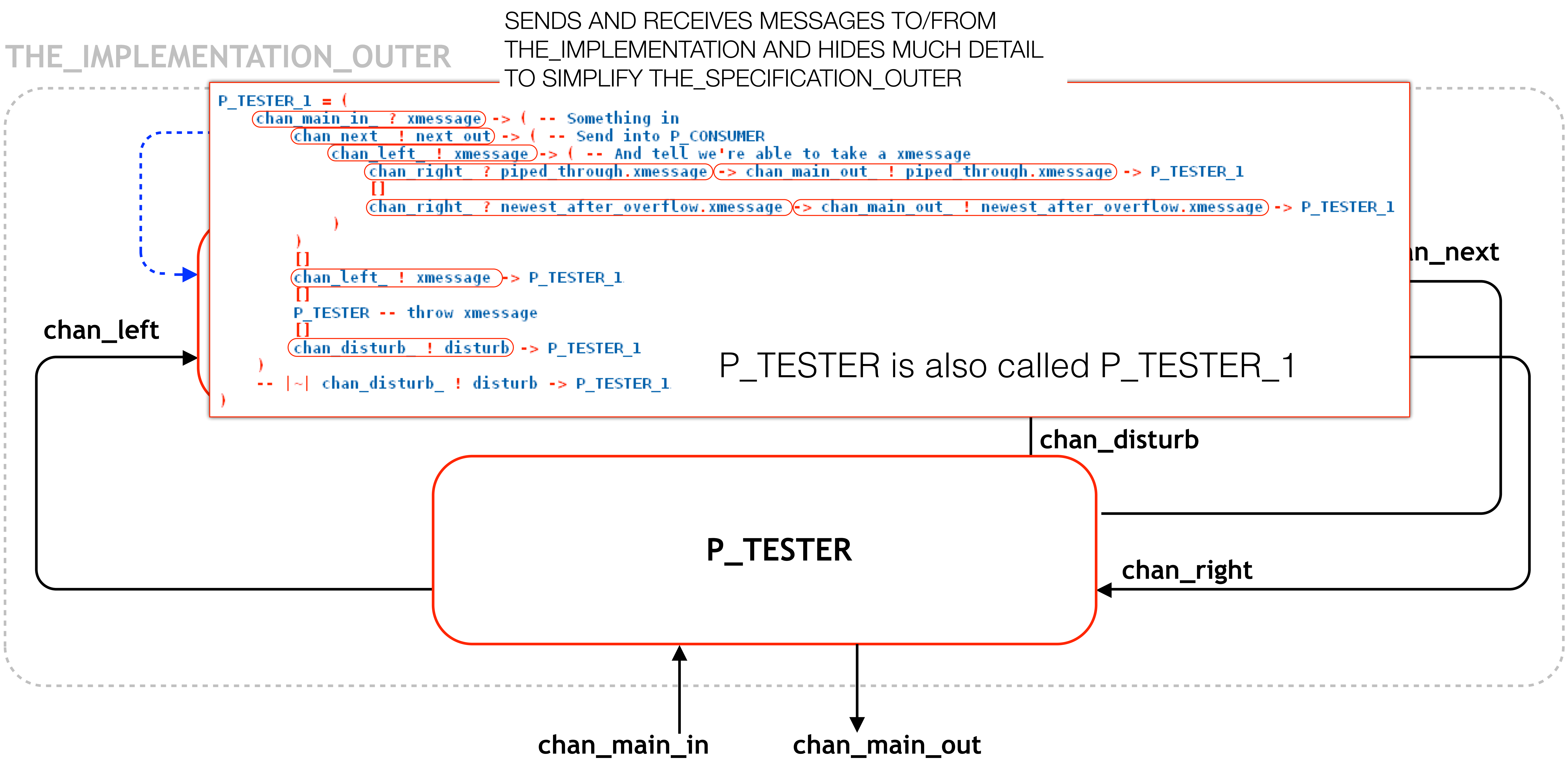

THE_IMPLEMENTATION_OUTER
SENDS AND RECEIVES MESSAGES TO/FROM THE_IMPLEMENTATION AND HIDES MUCH DETAIL TO SIMPLIFY THE_SPECIFICATION_OUTER
P_TESTER_1 = (
chan_main_in_ ? xmessage -> (-- Something in
chan_next_ ! next_out -> (-- Send into P_CONSUMER
chan_left_ ! xmessage -> (-- And tell we're able to take a xmessage
chan_right_ ? piped_through.xmessage -> chan_main_out_ ! piped_through.xmessage -> P_TESTER_1
[]
chan_right_ ? newest_after_overflow.xmessage -> chan_main_out_ ! newest_after_overflow.xmessage -> P_TESTER_1
)
)
[]
chan_left_ ! xmessage -> P_TESTER_1
[]
P_TESTER -- throw xmessage
[]
chan_disturb_ ! disturb -> P_TESTER_1
)
-- |~| chan_disturb_ ! disturb -> P_TESTER_1
)
P_TESTER is also called P_TESTER_1
chan_left
n_next
chan_disturb
P_TESTER
chan_right
chan_main_in
chan_main_out

Figure 4

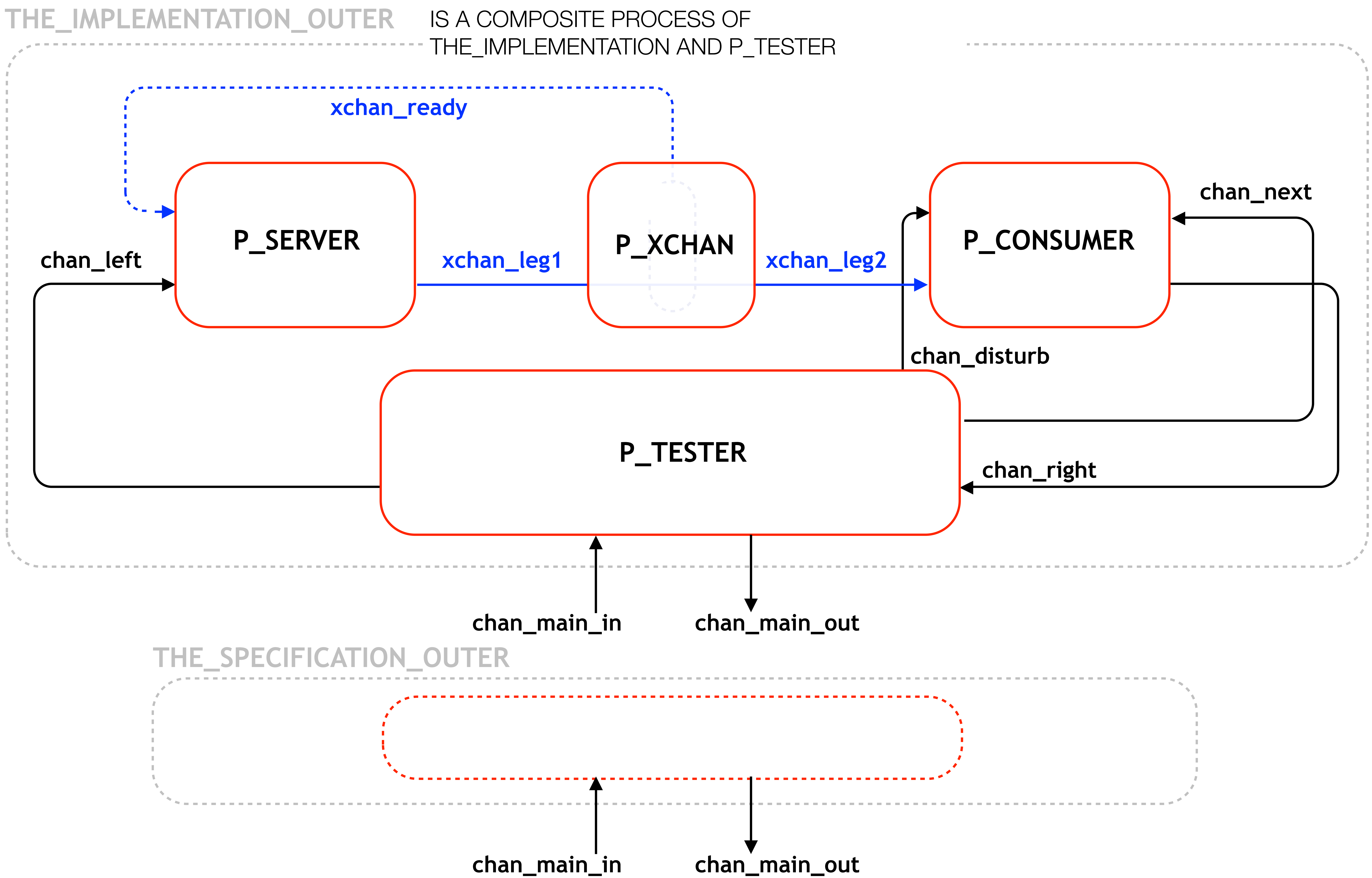
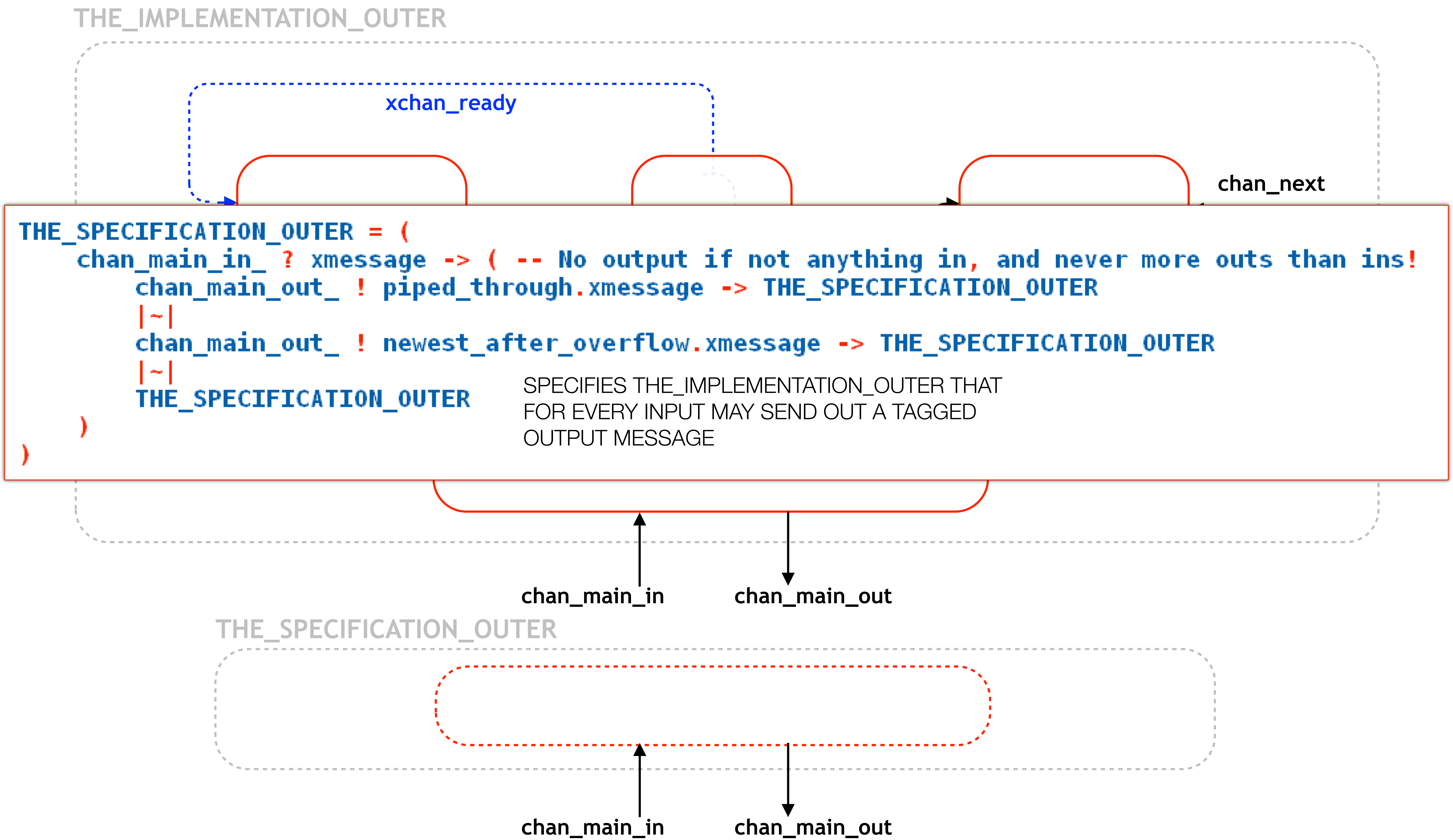

THE_IMPLEMENTATION_OUTER
IS A COMPOSITE PROCESS OF THE_IMPLEMENTATION AND P_TESTER
xchan_ready
P_SERVER
P_XCHAN
P_CONSUMER
chan_next
chan_left
xchan_leg1
xchan_leg2
chan_disturb
P_TESTER
chan_right
chan_main_in
chan_main_out
THE_SPECIFICATION_OUTER
chan_main_in
chan_main_out

Figure 5




```
THE_SPECIFICATION_OUTER = (
    chan_main_in_ ? xmessage -> ( -- No output if not anything in, and never more outs than ins!
        chan_main_out_ ! piped_through.xmessage -> THE_SPECIFICATION_OUTER
        |~|
        chan_main_out_ ! newest_after_overflow.xmessage -> THE_SPECIFICATION_OUTER
        |~|
        THE_SPECIFICATION_OUTER
    )
)
```

SPECIFIES THE_IMPLEMENTATION_OUTER THAT
FOR EVERY INPUT MAY SEND OUT A TAGGED
OUTPUT MESSAGE

Figure 5

```
THE_SPECIFICATION_OUTER = (
    chan_main_in_ ? xmessage -> ( -- No output if not anything in, and never more outs than ins!
        chan_main_out_ ! piped_through.xmessage -> THE_SPECIFICATION_OUTER
        |~|
        chan_main_out_ ! newest_after_overflow.xmessage -> THE_SPECIFICATION_OUTER
        |~|
        THE_SPECIFICATION_OUTER
    )
)
```

SPECIFIES THE_IMPLEMENTATION_OUTER THAT FOR EVERY INPUT MAY SEND OUT A TAGGED OUTPUT MESSAGE

What value (if any!) does such a general specification have?

```
THE_SPECIFICATION_OUTER = (
    chan_main_in_ ? xmessage -> ( -- No output if not anything in, and never more outs than ins!
        chan_main_out_ ! piped_through.xmessage -> THE_SPECIFICATION_OUTER
        |~|
        chan_main_out_ ! newest_after_overflow.xmessage -> THE_SPECIFICATION_OUTER
        |~|
        THE_SPECIFICATION_OUTER
    )
)
```

SPECIFIES THE_IMPLEMENTATION_OUTER THAT FOR EVERY INPUT MAY SEND OUT A TAGGED OUTPUT MESSAGE

«Different from LTL assertions, an assertion for refinement compares the whole behaviors of a given process with another process, e.g., whether there is a subset relationship» (11)

Hands on: deadlock

1. Introduction
2. Theory: XCHAN
3. **Hands on: deadlock**
4. Determinism-analysis of the XCHAN model
5. Conclusion

Refusals and acceptances

Refusals

What events a state *may* not engage in

Acceptances

What events a state *must* engage in,
if its environment desires

The one is the complement of the other «in Σ»

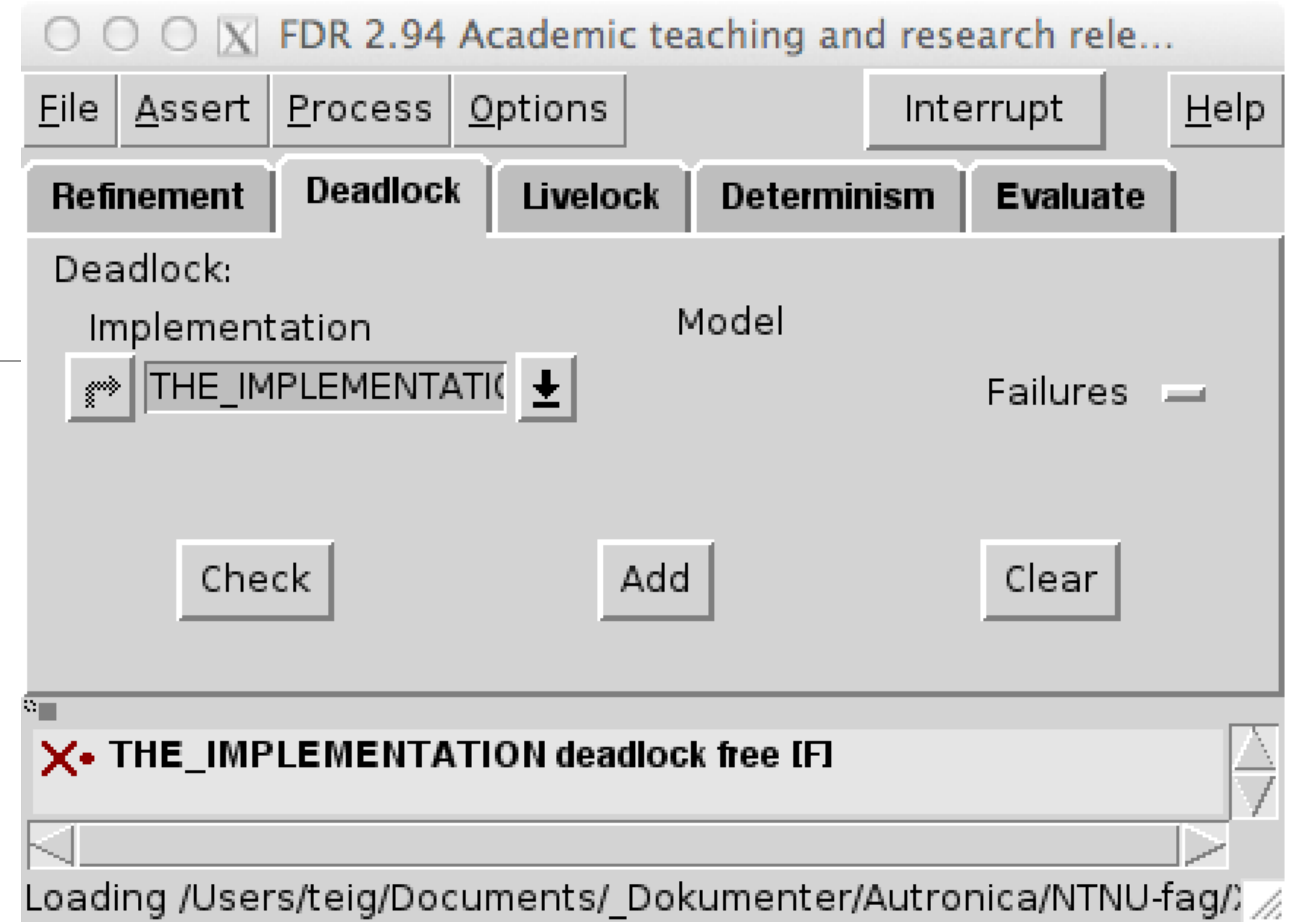
(6) p38

Deadlock: FDR2

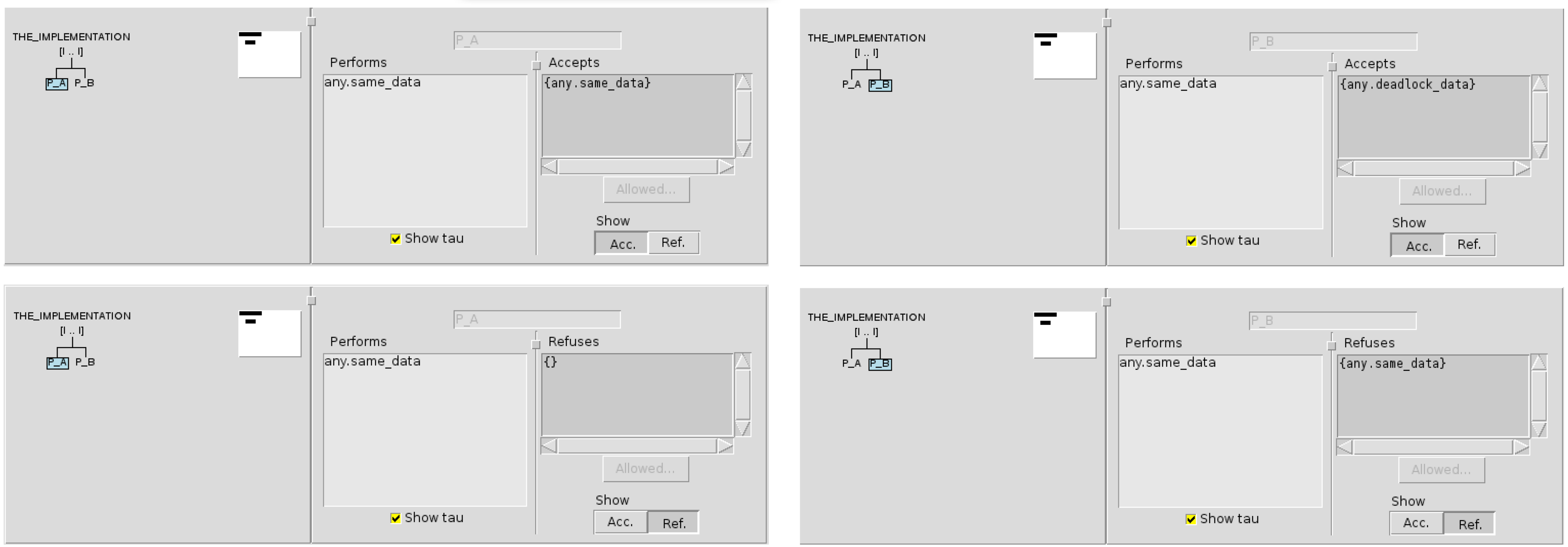
2013-03-06-004-no-deadlock-then-deadlock.csp

```
datatype data = same_data | deadlock_data -- 'data' == 'event'
channel any: data

P_A = (any ! same_data -> any ! same_data -> P_A)

P_B = (any ! same_data -> any ! deadlock_data -> P_B)

THE_IMPLEMENTATION = (P_A [| {| any |} |] P_B)
```

```
About to start deadlock check
Refinement check: *
+.*
+.Refusal error after 2 states

Refine checked 2 states
With 1 transitions

Found 1 example
Took 0(0+0) seconds
```



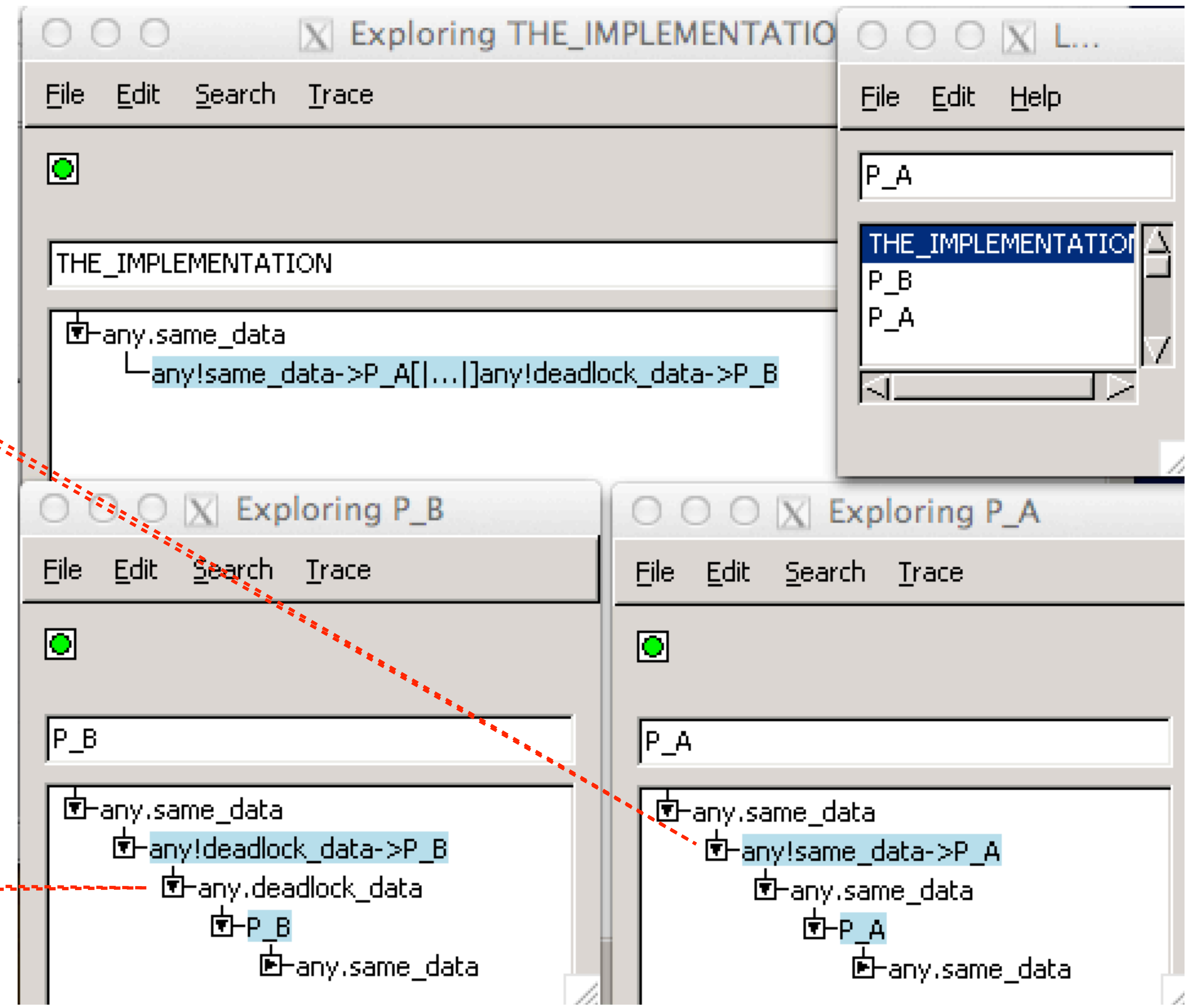
occam deadlocked here because **!** has semantic meaning, in FDR2 it's only syntactic sugar for `any.same_data` without direction!



Deadlock: ProBE

2013-03-06-004-no-deadlock-then-deadlock.csp

```
datatype data = same_data | deadlock_data -- 'data' == 'event'
channel any: data

P_A = (any ! same_data -> any ! same_data -> P_A)

P_B = (any ! same_data -> any ! deadlock_data -> P_B)

THE_IMPLEMENTATION = (P_A [| {| any |} |] P_B)
```




```
P_A = (any.same_data -> any.same_data -> P_A)

P_B = (any.same_data -> any.deadlock_data -> P_B)
```

Deadlock and hiding

Hiding *can* introduce divergence, and therefore invalidate many failures/divergences model specifications

In the stable failures model, a system P can deadlock if and only if $P \backslash \Sigma$ can. In other words, *we can hide absolutely all events — and move this hiding as far into the process as possible* using the principles already discussed

(6) p35-36

Determinism-analysis of the XCHAN model

1. Introduction
2. Theory: XCHAN
3. Hands on: deadlock
4. **Determinism-analysis of the XCHAN model**
5. Conclusion

Simply because

- I struggled more with this than with anything else

Pick one and satisfy, but find the right one(s)?

4.3 Choice of Model

The hierarchy of models for CSP are useful because they provide differing amount of information about the processes, with a corresponding change in the cost of working in that model. **It is more efficient to perform a check in the simplest model which provides the required detail.** (FDR2 (6) manual page 33)

	Property	Model	CSPm	assert # here
«Simple»	Safety STOP'ed train = fine!	Traces (refinement) Do not know what will happen! STOP refines all!	[T	4, 10
	Liveness Deadlock-freedom Determinism	Failures (refinement) Constrains what it is permitted to block and perform	[F	5, 11
«Complex»	Livelock-freedom Liveness properties Safety also here Deadlock also here Determinism also here	Failures-divergence (refinement) After *divergence trace*, then livelock (CHAOS). Detect livelock and used actively to make events not visible, hidden	[FD :[livelock free] :[deadlock free] :[deterministic]	6, 12 2, 8 1, 7 3, 9

FDR2 «allows the automatic checking of deadlock and livelock freedom as well as general safety and liveness properties» (10)

In *words*

Safety [T

- «There should never be a train and a car on the cross point at the same time» (10)
- XCHAN
  - ✓ «A message shall never be lost in XCHAN if there is an available receiver, on a message-per-message basis»
  - ✓ «Over time a fast producer and slow consumer may cause messages to become lost. The XCHAN sending side (application layer like P_SERVER) is in full control to take whatever action it wants to ensure that the required safety level is upheld.»

Liveness [F

- «Whenever a car or a train approaches the crossing they should eventually be able to cross» (10)
- XCHAN
  - ✓ «If buffer capacity is reached and no more data arrives all data will eventually be available for a receiver»

The CSPm *requirement* and *model* should then reflect this

- We can't just write *anything* and then press *any button* to verify that a requested property holds, like for any other sw program

- However, FDR2 (or I) will pick from its chest of tools whenever I have written some CSPm and I press the Check button

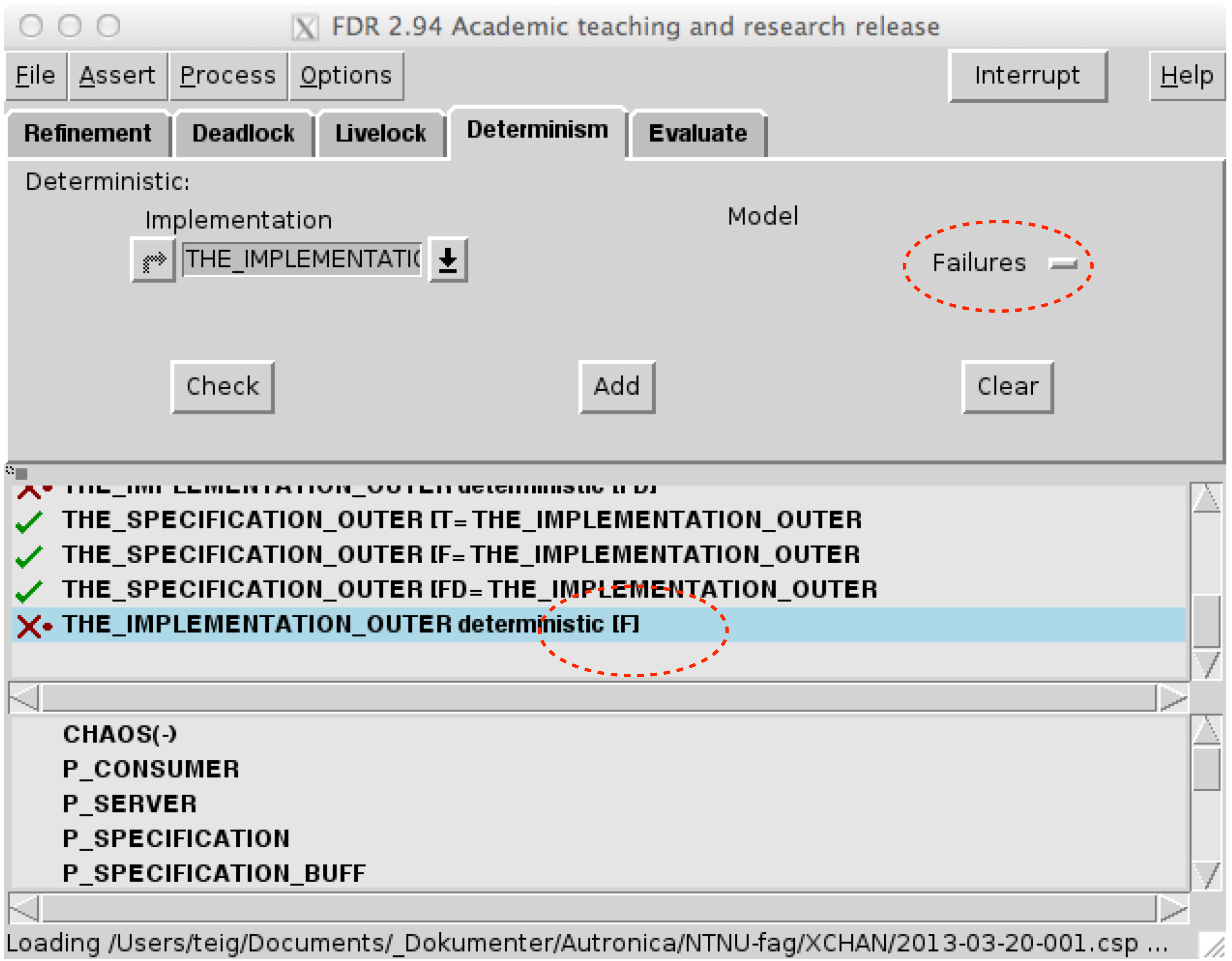
- I will then have the «determinism property» of the (good or bad) model I have written verified

- Remember that STOP satisfies any safety specification (like a trian that stands still) and that STOP is the simplest deadlocked process

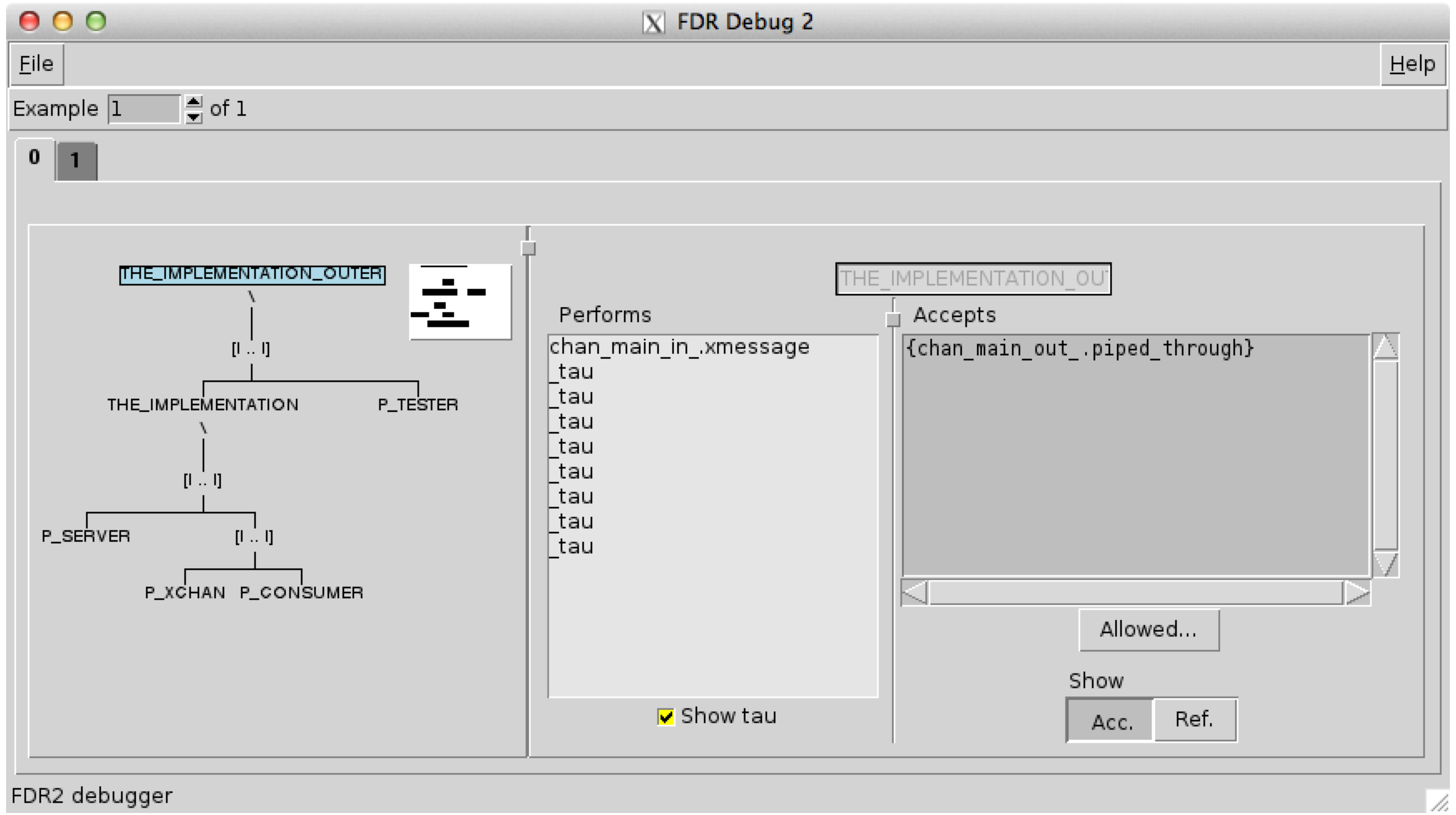
- Therefore we use several properties to tick off as verified the required properties. This sum of the results proves the final system

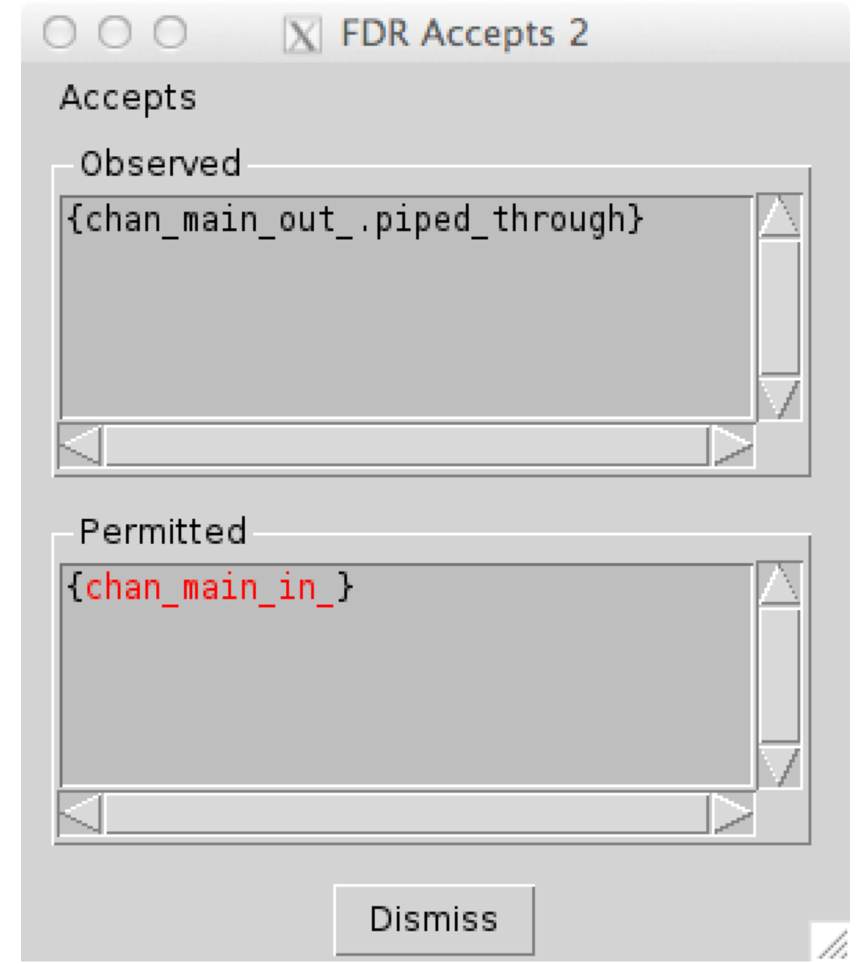
Understanding X·THE_IMPLEMENTATION_OUTER deterministic[F]

```
assert THE_IMPLEMENTATION       :[deadlock free]          -- #01 ok : deadlock property [FD]
assert THE_IMPLEMENTATION       :[livelock free]          -- #02 ok : livelock property
assert THE_IMPLEMENTATION       :[deterministic]          -- #03 err: deterministic     [F]
assert THE_SPECIFICATION         [T=  THE_IMPLEMENTATION  -- #04 err: safety property (FDR2 man p33..:)
assert THE_SPECIFICATION         [F=  THE_IMPLEMENTATION  -- #05 err: liveness or deadlock-freedom properties
assert THE_SPECIFICATION         [FD= THE_IMPLEMENTATION  -- #06 err: livelock-freedom property

assert THE_IMPLEMENTATION_OUTER :[deadlock free]          -- #07 ok : deadlock property [FD]
assert THE_IMPLEMENTATION_OUTER :[livelock free]          -- #08 ok : livelock property
assert THE_IMPLEMENTATION_OUTER :[deterministic]          -- #09 err: deterministic     [F]
assert THE_SPECIFICATION_OUTER   [T=  THE_IMPLEMENTATION_OUTER -- #10 ok : safety property (FDR2 man p33..:)
assert THE_SPECIFICATION_OUTER   [F=  THE_IMPLEMENTATION_OUTER -- #11 ok : liveness or deadlock-freedom properties
assert THE_SPECIFICATION_OUTER   [FD= THE_IMPLEMENTATION_OUTER -- #12 ok : livelock-freedom property
```

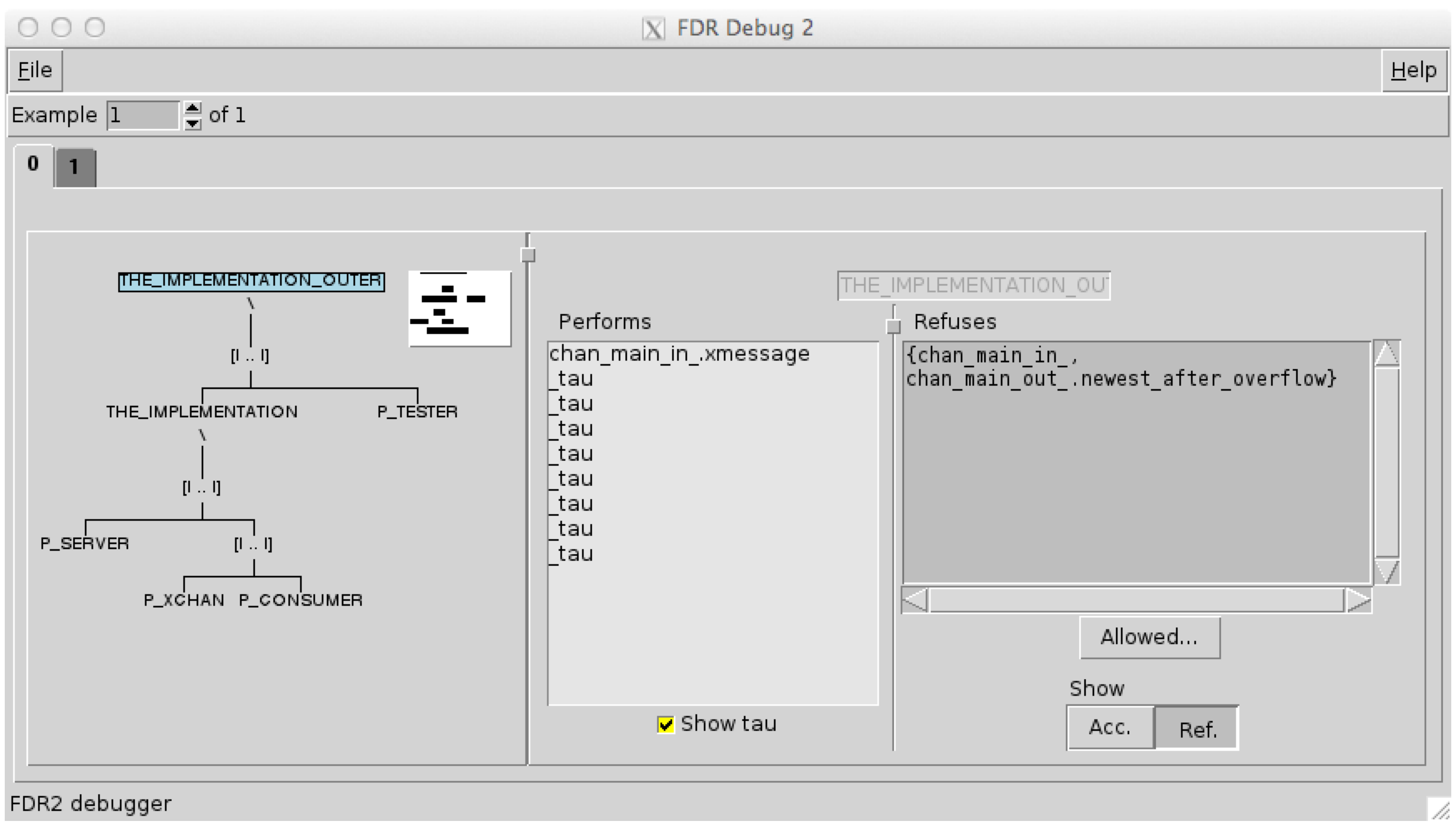


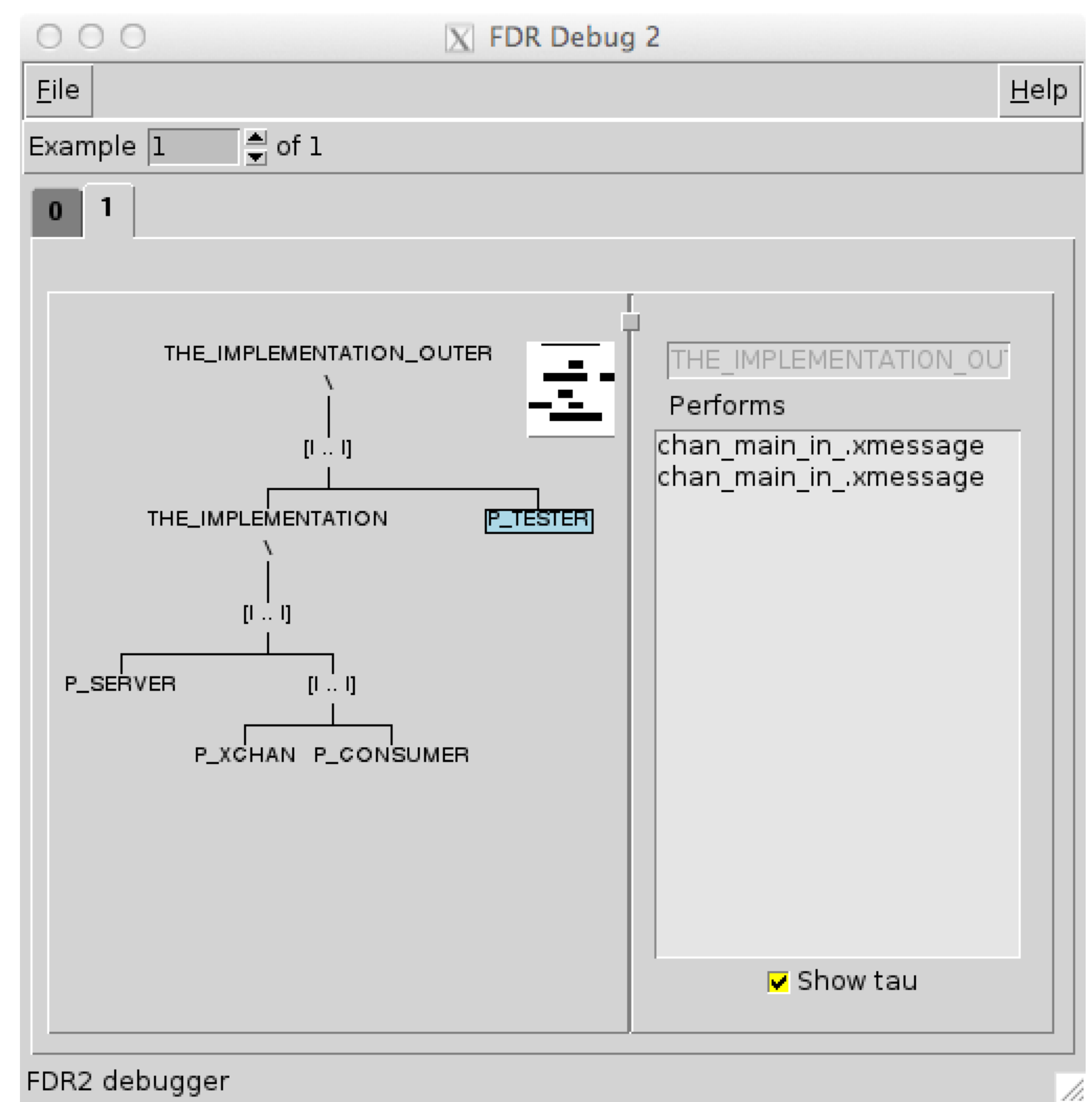
FDR2



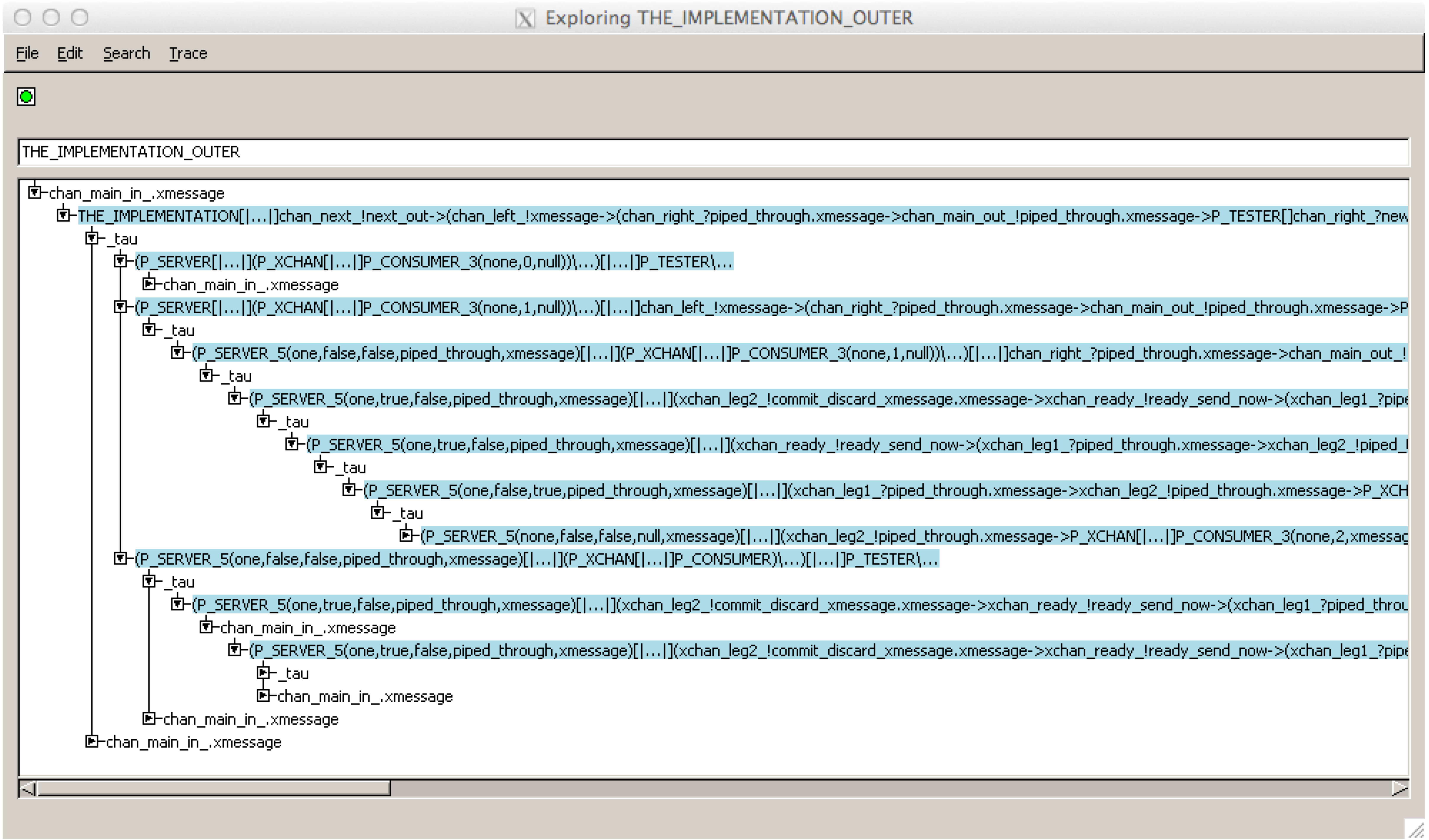


«In this case, it is a failure of liveness, which you can tell by the right-hand area having the heading Accepts. Such a behaviour consists of a perfectly acceptable trace of events performed by THE_IMPLEMENTATION_OUTER and an unacceptably small set of events that THE_IMPLEMENTATION_OUTER may then offer to its environment.» (FDR2 manual p27 rewritten for this system)





ProBE



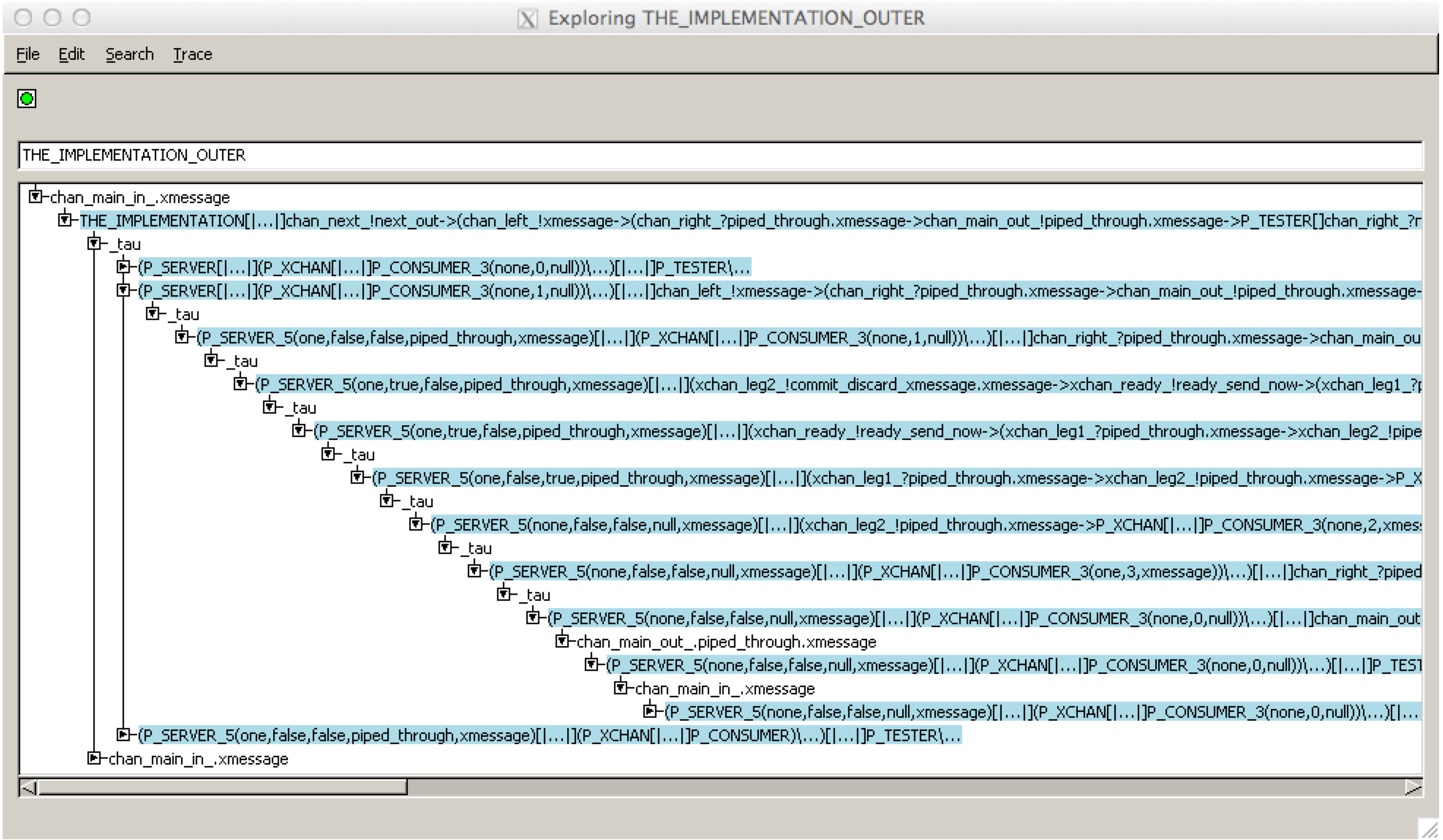
Observe that the ProBE diagram above jumps right into THE_IMPLEMENTATION and then into P_SERVER, not THE_IMPLEMENTATION_OUTER

ProBE

```
Level 2
THE_IMPLEMENTATION[|...|]chan_next_!next_out->(chan_left_!xmessage->(chan_right_?piped_through.xmessage->chan_main_out_!piped_through.xmessage->P_TESTER[]chan_right_?newest_after_overflow.xmessage->chan_main_out_!newest_after_overflow.xmessage->P_TESTER))[]chan_left_!xmessage->P_TESTER[]P_TESTER[]chan_disturb_!disturb->P_TESTER\...

Level 3
(P_SERVER[|...|](P_XCHAN[|...|]P_CONSUMER_3(none,1,null))\...)[|...|]chan_left_!xmessage->(chan_right_?piped_through.xmessage->chan_main_out_!piped_through.xmessage->P_TESTER[]chan_right_?newest_after_overflow.xmessage->chan_main_out_!newest_after_overflow.xmessage->P_TESTER)\...

Level 4
(P_SERVER_5(one,false,false,piped_through,xmessage)[|...|](P_XCHAN[|...|]P_CONSUMER_3(none,1,null))\...)[|...|]chan_right_?piped_through.xmessage->chan_main_out_!piped_through.xmessage->P_TESTER[]chan_right_?newest_after_overflow.xmessage->chan_main_out_!newest_after_overflow.xmessage->P_TESTER\...

Level 5
(P_SERVER_5(one,true,false,piped_through,xmessage)[|...|](xchan_leg2_!commit_discard_xmessage.xmessage->xchan_ready_!ready_send_now->(xchan_leg1_?piped_through.xmessage->xchan_leg2_!piped_through.xmessage->P_XCHAN[]xchan_leg1_?newest_after_overflow.xmessage->xchan_leg2_!newest_after_overflow.xmessage->P_XCHAN)[|...|]P_CONSUMER_3(none,1,null))\...)[|...|]chan_right_?piped_through.xmessage->chan_main_out_!piped_through.xmessage->P_TESTER[]chan_right_?newest_after_overflow.xmessage->chan_main_out_!newest_after_overflow.xmessage->P_TESTER\...

THIS IS level 5:
```


```
Level 6
(P_SERVER_5(one,true,false,piped_through,xmessage)[|...|](xchan_ready_!ready_send_now->(xchan_leg1_?piped_through.xmessage->xchan_leg2_!piped_through.xmessage->P_XCHAN[]xchan_leg1_?newest_after_overflow.xmessage->xchan_leg2_!newest_after_overflow.xmessage->P_XCHAN)[|...|]P_CONSUMER_3(none,2,xmessage))\...)[|...|]chan_right_?piped_through.xmessage->chan_main_out_!piped_through.xmessage->P_TESTER[]chan_right_?newest_after_overflow.xmessage->chan_main_out_!newest_after_overflow.xmessage->P_TESTER\...

Level 7
(P_SERVER_5(one,false,true,piped_through,xmessage)[|...|](xchan_leg1_?piped_through.xmessage->xchan_leg2_!piped_through.xmessage->P_XCHAN[]xchan_leg1_?newest_after_overflow.xmessage->xchan_leg2_!newest_after_overflow.xmessage->P_XCHAN[|...|]P_CONSUMER_3(none,2,xmessage))\...)[|...|]chan_right_?piped_through.xmessage->chan_main_out_!piped_through.xmessage->P_TESTER[]chan_right_?newest_after_overflow.xmessage->chan_main_out_!newest_after_overflow.xmessage->P_TESTER\...
```

ProBE



So, this is not the trace, is it..? But we discuss no-determinisn here..? Hmm.

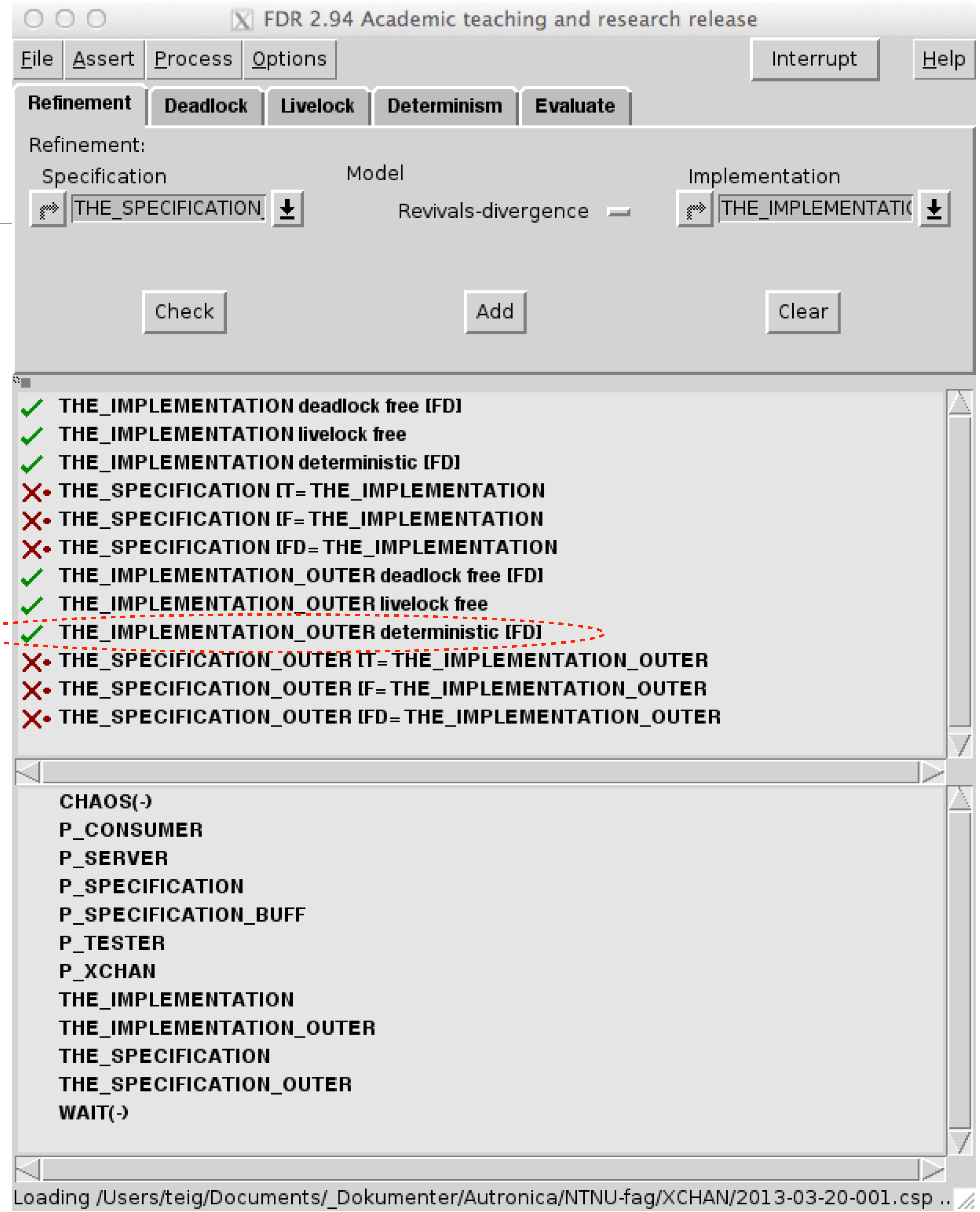
Experimenting with hiding

experiment-1:
removing three hidings will make it deterministic, but fail others!
Search for «experiment-1» in the «2013-03-20-001.csp» file.
It makes both property sets (above and below red line) equal

```
THE_IMPLEMENTATION = (
   ...
-- \ {| xchan_ready_, xchan_leg1_, xchan_leg2_ |}
-- experiment-1: to get it deterministic: remove hiding here (1/3)

THE_SPECIFICATION = (
    P_SPECIFICATION_BUFF
    [| {|chan_mid_|} |]
    P_SPECIFICATION)
-- \ {| chan_mid_ |}
-- experiment-1: to get it deterministic: remove hiding here (2/3)

THE_IMPLEMENTATION_OUTER = (
    ...
-- \ {| chan_left_, chan_right_, chan_next_, chan_disturb_ |}
-- experiment-1: to get it deterministic: remove hiding here (3/3)
```

Generalized parallel /
interface parallel / sharing

Hiding makes things less obvious and opens for surprises - so determinism may fail because of this!



FDR2 in batch mode. Trail 1

FDR2 batch -trace -depth 5 -refusals /Users/teig/Documents/_Dokumenter/Autronica/NTNU-fag/XCHAN/2013-03-20-001.csp

If -trace has been selected, then report traces for sub-processes as well as the root processes. This is the same as expanding the specified number of levels of the tree in the FDR debugger, noting down the traces for each sub-process. The BEGIN TRACE/ END TRACE lines carry additional information indicating the path through from the root to the sub-process which generate the particular trace (6)

A typical use of -depth is when the CSP script uses hiding and compression and extracting the full counter-example requires 'tunneling' inside those sub-processes. This is often the case when the CSP has been automatically generated from some other notation.

FDR2 produces 6 «trails» for me. I have named them Trail:1 to Trail: 6. 5-6 not listed here (space).

```
(BEGIN batch -depth 5)

Checking THE_IMPLEMENTATION_OUTER :[deterministic]
Starting timer
Starting compilation

Starting...
Compiling...
Reading...
 Loading... done
Took 0(0+0) seconds
Starting timer
About to start determinism check
Allocated a total of 2 pages of size 128K
Compaction produced 0 chunks of 16K.
Refinement check:
Trace error after 2 states
Refine checked 2 states
With 1 transitions

Found 1 example
Took 0(0+0) seconds
Refinement check:
Refusal error after 16 states
Refine checked 16 states
With 16 transitions
Allocated a total of 8 pages of size 128K
Compaction produced 0 chunks of 16K.
xfalse
BEGIN BEHAVIOUR example=0 process=0 path=0
```

```
BEGIN TRACE (Trail:1)
chan_main_in_.xmessage
_tau
_tau
_tau
_tau
_tau
_tau
_tau
_tau
END TRACE
BEGIN ACCEPTANCES
chan_main_out_.piped_through
END ACCEPTANCES
BEGIN REFUSALS
chan_main_in_
chan_main_out_.newest_after_overflow
END REFUSALS
END BEHAVIOUR example=0 process=0 path=0

BEGIN BEHAVIOUR example=0 process=0 path=0 0
```

Trail 2-4 (5-6 not shown)

```
BEGIN TRACE (Trail: 2)
chan_main_in_.xmessage
chan_next_.next_out
chan_left_.xmessage
_tau
_tau
_tau
_tau
_tau
chan_right_.piped_through.xmessage
END TRACE
BEGIN ACCEPTANCES
chan_main_out_.piped_through
END ACCEPTANCES
BEGIN REFUSALS
chan_disturb_
chan_left_
chan_main_in_
chan_main_out_.newest_after_overflow
chan_next_
chan_right_.newest_after_overflow
chan_right_.piped_through
END REFUSALS
END BEHAVIOUR example=0 process=0 path=0 0

BEGIN BEHAVIOUR example=0 process=0 path=0 0 0
```

```
BEGIN TRACE (Trail: 3)
chan_next_.next_out
chan_left_.xmessage
_tau
_tau
_tau
_tau
_tau
chan_right_.piped_through.xmessage
END TRACE
BEGIN ACCEPTANCES
chan_disturb_
chan_left_
chan_next_
END ACCEPTANCES
BEGIN REFUSALS
chan_right_.newest_after_overflow
chan_right_.piped_through
END REFUSALS
END BEHAVIOUR example=0 process=0 path=0 0 0

BEGIN BEHAVIOUR example=0 process=0 path=0 0 0 0
```

```
BEGIN TRACE (Trail: 4)
chan_next_.next_out
chan_left_.xmessage
xchan_ready_.ready_sender_has_xmessage
xchan_leg2_.commit_discard_xmessage.xmessage
xchan_ready_.ready_send_now
xchan_leg1_.piped_through.xmessage
xchan_leg2_.piped_through.xmessage
chan_right_.piped_through.xmessage
END TRACE
BEGIN ACCEPTANCES
chan_disturb_
chan_left_
chan_next_
END ACCEPTANCES
BEGIN REFUSALS
chan_right_.newest_after_overflow
chan_right_.piped_through
xchan_leg1_.newest_after_overflow
xchan_leg1_.piped_through
xchan_leg2_
xchan_ready_
END REFUSALS
END BEHAVIOUR example=0 process=0 path=0 0 0 0

BEGIN BEHAVIOUR example=0 process=0 path=0 0 0 0 0
```

Traces, acceptances and refusals tables

TRACE of THE_IMPLEMENTATION_OUTER :[deterministic]

Trail: 1	Trail: 2	Trail: 3	Trail: 4	Trail: 5	Trail: 6
chan_main_in_.xmessage	chan_main_in_.xmessage				
_tau	chan_next_.next_out	chan_next_.next_out	chan_next_.next_out		chan_next_.next_out
_tau	chan_left_.xmessage	chan_left_.xmessage	chan_left_.xmessage	chan_left_.xmessage	
_tau	_tau	_tau	xchan_ready_.ready_sender_has_xmessage		xchan_ready_.ready_sender_has_xmessage
_tau	_tau	_tau	xchan_leg2_.commit_discard_xmessage.xmessage	xchan_ready_.ready_sender_has_xmessage	xchan_leg2_.commit_discard_xmessage.xmessage
_tau	_tau	_tau	xchan_ready_.ready_send_now	xchan_ready_.ready_send_now	xchan_ready_.ready_send_now
_tau	_tau	_tau	xchan_leg1_.piped_through.xmessage	xchan_leg1_.piped_through.xmessage	xchan_leg1_.piped_through.xmessage
_tau	_tau	_tau	xchan_leg2_.piped_through.xmessage		xchan_leg2_.piped_through.xmessage
_tau	chan_right_.piped_through.xmessage	chan_right_.piped_through.xmessage	chan_right_.piped_through.xmessage		chan_right_.piped_through.xmessage

ACCEPTANCES of THE_IMPLEMENTATION_OUTER :[deterministic]

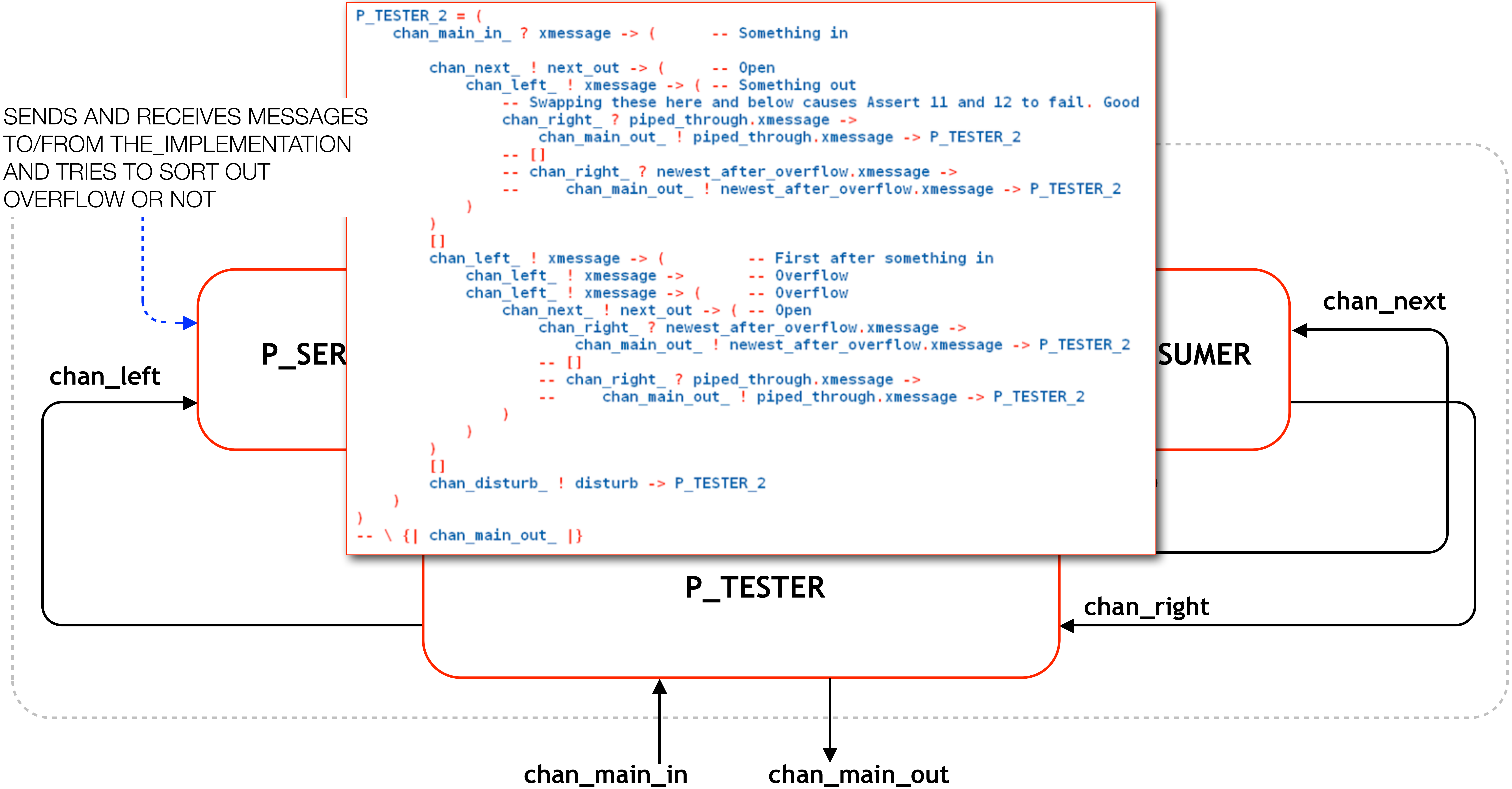
Trail: 1	Trail: 2	Trail: 3	Trail: 4	Trail: 5	Trail: 6
chan_main_out_.piped_through	chan_main_out_.piped_through				
		chan_disturb_	chan_disturb_		chan_disturb_
		chan_left_	chan_left_	chan_left_	
		chan_next_	chan_next_		chan_next_
					xchan_ready_.ready_sender_has_xmessage

REFUSALS of THE_IMPLEMENTATION_OUTER :[deterministic] (There is only external [] choice in use, still we have refusals...?)

Trail: 1	Trail: 2	Trail: 3	Trail: 4	Trail: 5	Trail:Trail: 6
	chan_disturb_				
	chan_left_				
chan_main_in_	chan_main_in_				
chan_main_out_.newest_after_overflow	chan_main_out_.newest_after_overflow				
	chan_next_				
	chan_right_.newest_after_overflow	chan_right_.newest_after_overflow	chan_right_.newest_after_overflow		chan_right_.newest_after_overflow
	chan_right_.piped_through	chan_right_.piped_through	chan_right_.piped_through		chan_right_.piped_through
			xchan_leg1_.newest_after_overflow	xchan_leg1_.newest_after_overflow	xchan_leg1_.newest_after_overflow
			xchan_leg1_.piped_through	xchan_leg1_.piped_through	xchan_leg1_.piped_through
			xchan_leg2_		xchan_leg2_
			xchan_ready_	xchan_ready_	
					xchan_ready_.ready_send_now

Drawn by hand

Nothing wrong here!

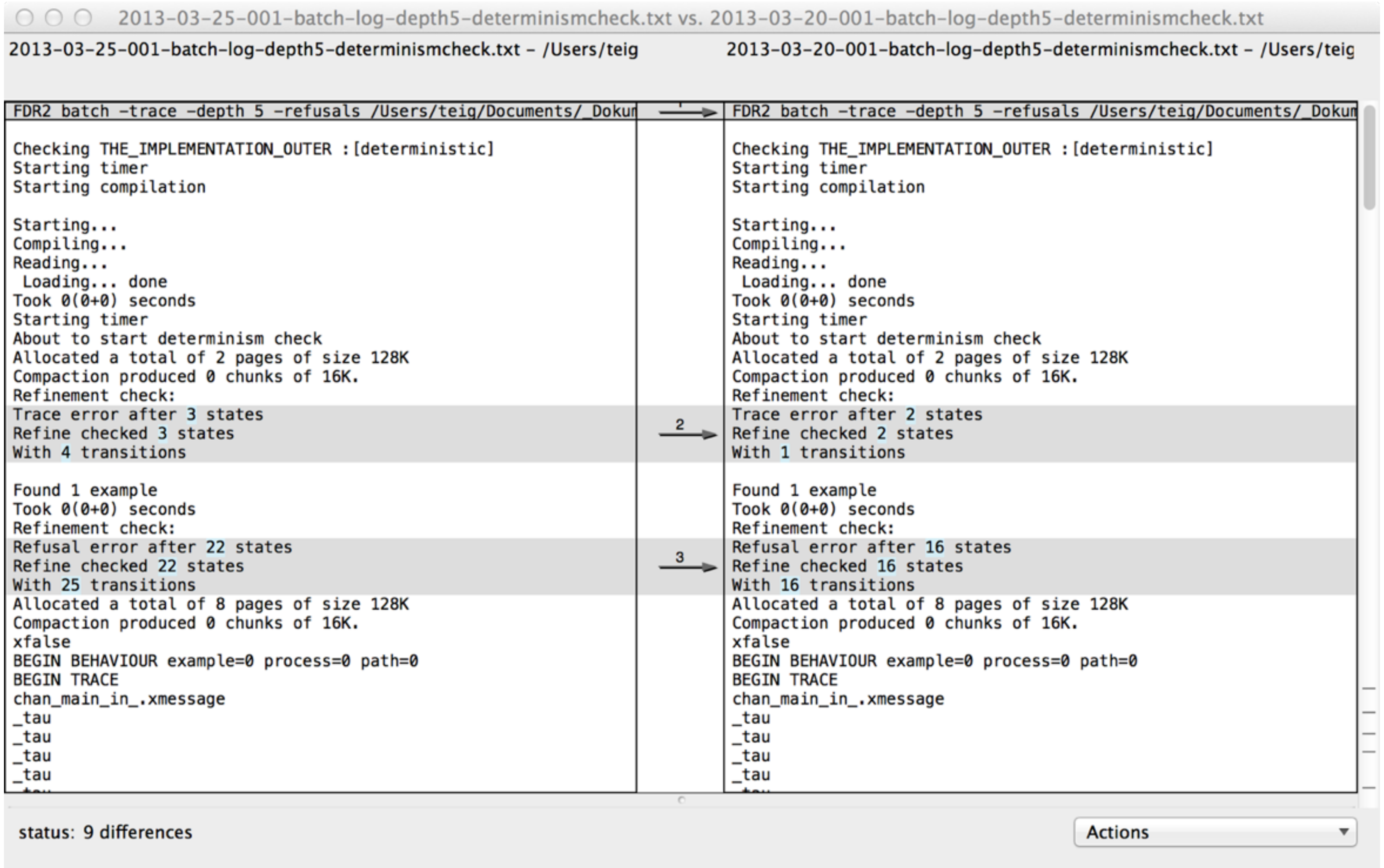



```
P_TESTER_2 = (
    chan_main_in_ ? xmessage -> (      -- Something in

        chan_next_ ! next_out -> (      -- Open
            chan_left_ ! xmessage -> ( -- Something out
                -- Swapping these here and below causes Assert 11 and 12 to fail. Good
                chan_right_ ? piped_through.xmessage ->
                    chan_main_out_ ! piped_through.xmessage -> P_TESTER_2
                -- []
                -- chan_right_ ? newest_after_overflow.xmessage ->
                --     chan_main_out_ ! newest_after_overflow.xmessage -> P_TESTER_2
            )
        )
        []
        chan_left_ ! xmessage -> (          -- First after something in
            chan_left_ ! xmessage ->        -- Overflow
            chan_left_ ! xmessage -> (      -- Overflow
                chan_next_ ! next_out -> ( -- Open
                    chan_right_ ? newest_after_overflow.xmessage ->
                        chan_main_out_ ! newest_after_overflow.xmessage -> P_TESTER_2
                    -- []
                    -- chan_right_ ? piped_through.xmessage ->
                    --     chan_main_out_ ! piped_through.xmessage -> P_TESTER_2
                )
            )
        )
        []
        chan_disturb_ ! disturb -> P_TESTER_2
    )
)
-- \ {| chan_main_out_ |}
```

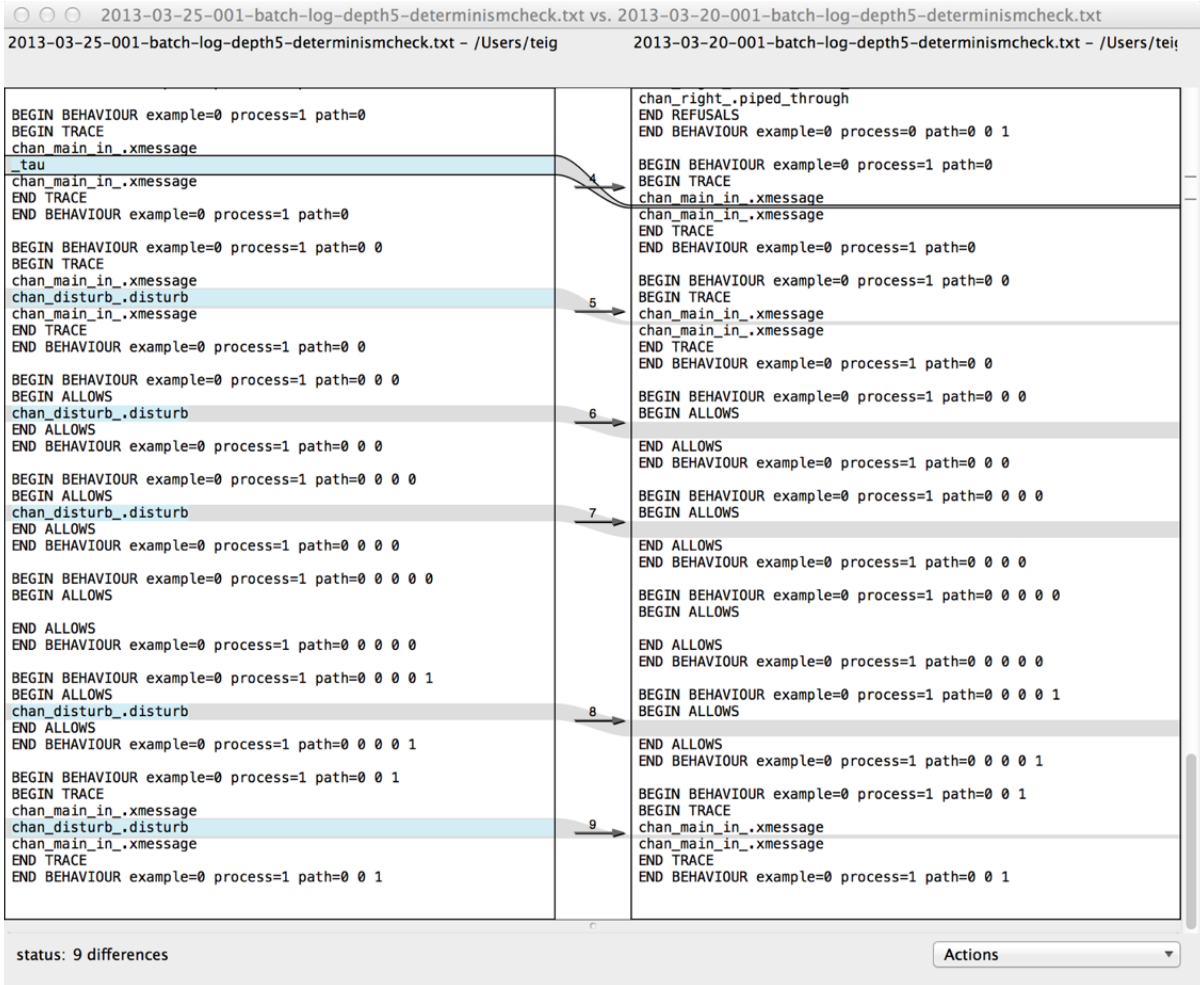
Figure 4

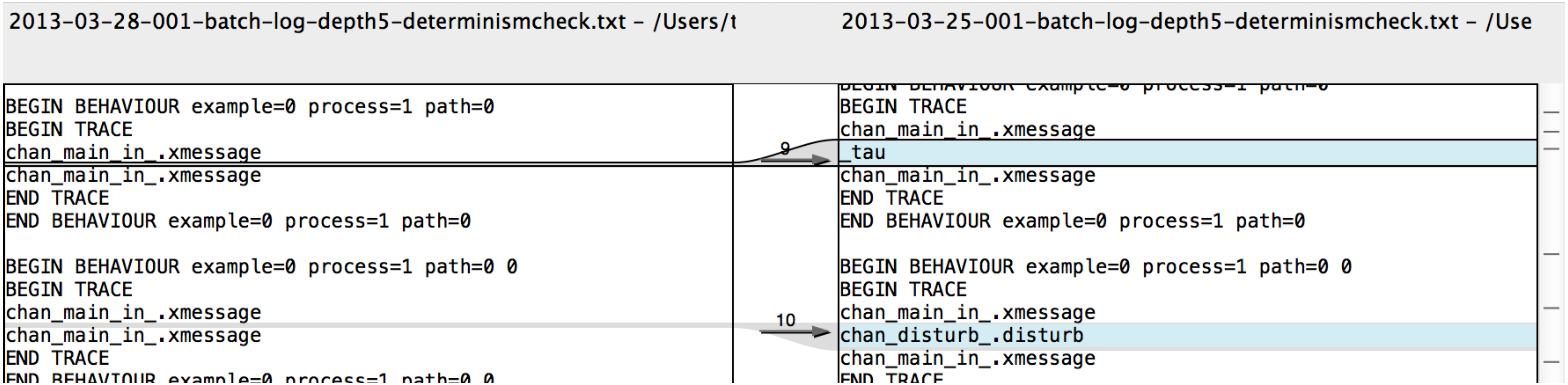
At this point this yields the same result as with see of the original P_TESTER

Diff’ing logs may be a good idea



But traces only differ on *_tau* and *disturb_*:






```
THE_IMPLEMENTATION_OUTER = (

    THE_IMPLEMENTATION

    [| {| chan_left_, chan_right_, chan_next_ |} |]

    P_TESTER_1 -- or P_TESTER_2
)
\ {| chan_left_, chan_right_, chan_next_ |}
```

```
THE_IMPLEMENTATION_OUTER = (

    THE_IMPLEMENTATION

    [| {| chan_left_, chan_right_, chan_next_, chan_disturb_ |} |]

    P_TESTER_2 -- or P_TESTER_1
)
\ {| chan_left_, chan_right_, chan_next_, chan_disturb_ |}
```

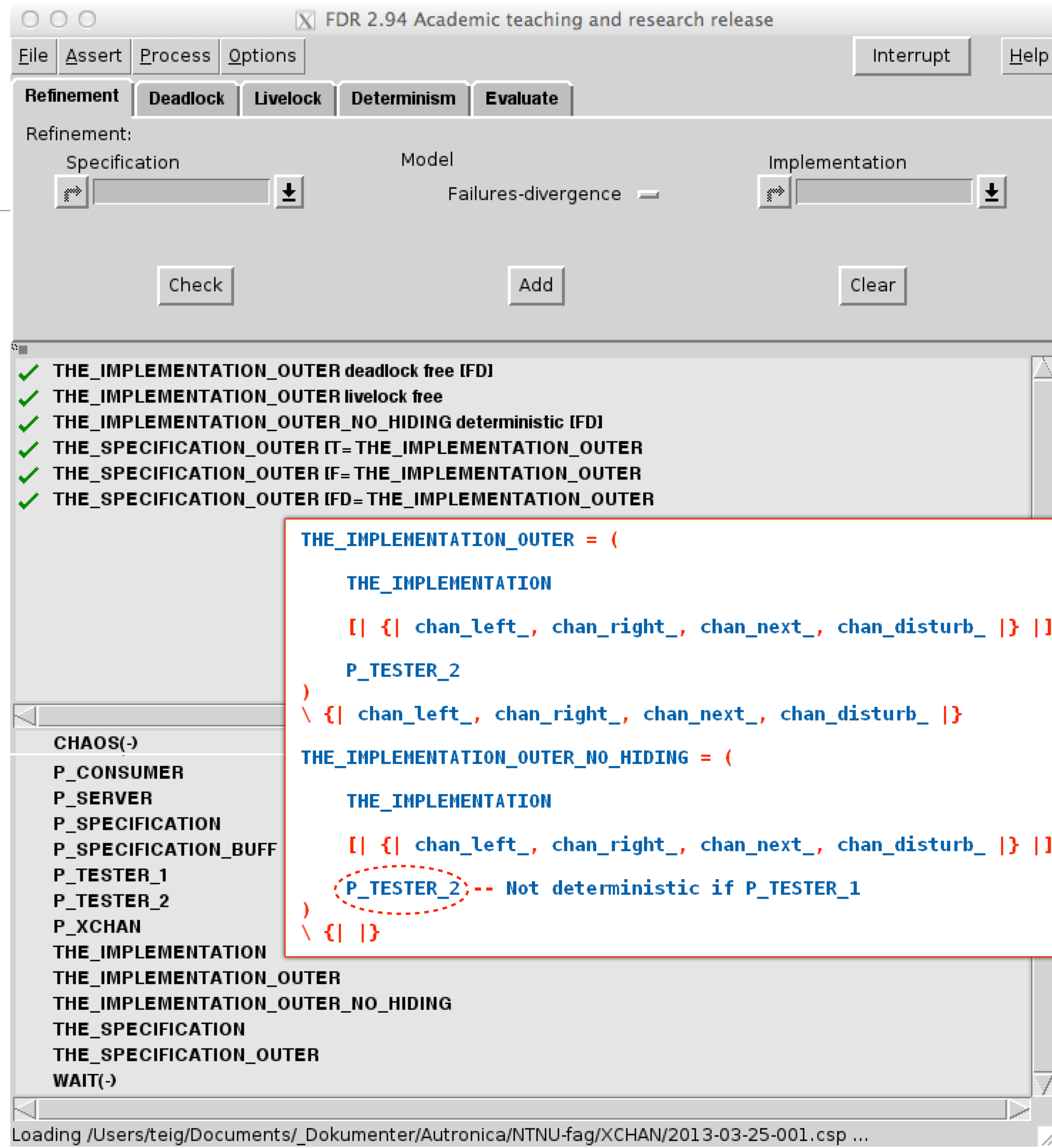
- I tried to remove chan_disturb_, but got the exact same result. Then also the _tau were gone, because chan_disturb_ was hidden in THE_IMPLEMENTATION_OUTER
- Same results with both P_TESTER_1 and P_TESTER_2
- This should indicate that chan_disturb is correctly modeled, since it in fact does not «disturb» at all

Finally..

- Removing hiding in THE_IMPLEMENTATION_OUTER made it deterministic!
- But only with the much more precise P_TESTER_2 which also relates to overflow
- QED?

```
P_TESTER_2 = (
    chan_main_in_ ? xmessage -> (       -- Something in
        chan_next_ ! next_out -> (      -- Open
            chan_left_ ! xmessage -> ( -- Something out
                chan_right_ ? piped_through.xmessage ->
                chan_main_out_ ! piped_through.xmessage ->
                P_TESTER_2
            )
        )
        []
        chan_left_ ! xmessage -> (          -- First after something in
            chan_left_ ! xmessage ->        -- Overflow
            chan_left_ ! xmessage -> (      -- Overflow
                chan_next_ ! next_out -> ( -- Open
                    chan_right_ ? newest_after_overflow.xmessage ->
                    chan_main_out_ ! newest_after_overflow.xmessage ->
                    P_TESTER_2
                )
            )
        )
        []
        chan_disturb_ ! disturb -> P_TESTER_2
    )
)
```

SENDS AND RECEIVES MESSAGES TO/FROM THE_IMPLEMENTATION AND TRIES TO SORT OUT OVERFLOW OR NOT




```
THE_IMPLEMENTATION_OUTER = (
    THE_IMPLEMENTATION
    [| {| chan_left_, chan_right_, chan_next_, chan_disturb_ |} |]
    P_TESTER_2
)
\ {| chan_left_, chan_right_, chan_next_, chan_disturb_ |}
THE_IMPLEMENTATION_OUTER_NO_HIDING = (
    THE_IMPLEMENTATION
    [| {| chan_left_, chan_right_, chan_next_, chan_disturb_ |} |]
    P_TESTER_2 -- Not deterministic if P_TESTER_1
)
\ {| |}
```

Conclusion of non-determinism evaluation

- After much effort I finally found a way to see that my implementation is deterministic!
- From ProBE it also seems to do what I have told it to do
- Even if I know that nondeterminism «comes from» hiding I had to «tune» and go all the way described in this section
- Observe that I have used
  [] (external choice) in *all implementations* and
  |~| (internal or nondeterministic choice) only in the *specifications*

CHAOS, WAIT

- Seem to be part of any process set in FDR2. I don't know why
- They do not show up in ProBE
- «CHAOS» is a CSPm keyword, it can always choose to communicate or reject. It is «the most deterministic divergence-free process» (7)
- «WAIT» is not in CSPm. It simply is a delay operator
- Neither is «RUN» (seen in CSP book (12)). It is «the process that will deterministically perform any event» (7)

Which tool and when?

- When ok fulfillment of a property:
  - observe the assumed behaviour with ProBE
  - remove some hiding to watch internal details
- When error:
  - use FDR2 and ProBE together
  - play around with hiding (and renaming?)
  - run FDR2 in batch mode with «depth» parameter

Conclusion

1. Introduction
2. Theory: XCHAN
3. Hands on: deadlock
4. Determinism-analysis of the XCHAN model
5. **Conclusion**

Conclusions

1. CSPm (as CSP) has a steep learning curve. TK8112 covers the foundations of CSP, but CSPm seemed to me to be a more different game than I had envisaged
2. How to succeed with FDR2 installation was not so obvious. FDR2 on OSX needed X11 (XQuartz). ProBE runs on WineApp.app on OSX. Wrote blog note, see (5)
3. After having become somewhat familiar with FDR2 and ProBE I encountered to understand how (or if) I could specify and model XCHAN (8)
4. The present model took me quite far with an «occam in CSPm» approach. I feel reasonably assured that I have specified and implemented models of the real XCHAN. But this is in some respects the hardest bit: dragging onself from the marsh to solid ground
5. Of course I have only *scratched* the surface of CSP and CSPm
6. It takes time to understand the CSPm landscape, even if CSPm is a language to formally treat something as «simple» as state machines (or labeled transition diagrams)

For NTNU

1. I recommend the next curriculum to include exercises in CSPm. The FDR2 / CSPm User Manual (6) is packed with a *very interesting* language! I have shown a flavour of it here. Because I have (all minus a flavour) left to learn!

2. And also doing exercises in PAT, the Process Analysis Toolkit from the universities in Singapore and Nanyang (10). Its CSP# language also contains LTL (Linear Temporal Logic) and works with C# and Microsoft Contracts. Generates code (but not for this example, since synchronous channels)

3. I also recommend *more group work*, because it's hard to drag oneself by the hair

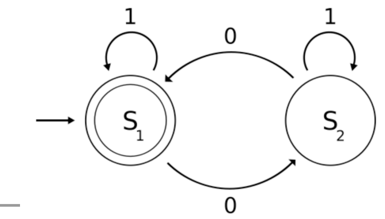
4. I must thank Sverre Hendseth, the lecturer, for his guidance and positive attitude

5. He certainly gave me the impression that there was not much prior work to draw on concerning CSPm, FDR2 and ProBE at NTNU

References

Becoming textual: attempting to model 'XCHAN' with CSPm : Using FDR2 and ProBE tools when state-ing is not enough

Øyvind Teig, Autronica Fire and Security : http://www.teigfam.net/oyvind/home/
Lecture material at: http://www.teigfam.net/oyvind/home/technology/063-lecture-ntnu/



(1) TTK3 - Sanntidsteori (NTNU), http://www.itk.ntnu.no/ansatte/Hendseth_Sverre/sanntidsteori/index.html

(2) TK8112 - The Theory of Concurrency in Real-Time Systems (NTNU), http://www.ntnu.edu/studies/courses/TK8112

(3) Formal Systems Europe, http://www.fsel.com

(4) University of Oxford, http://www.cs.ox.ac.uk/projects/concurrency-tools/ binaries for academic use

(5) «FDR2 notes», http://www.teigfam.net/oyvind/home/technology/057-fdr2-notes/ by Øyvind Teig.
It also contains some theory clarifications.

(6) FDR2 User Manual. Download from http://www.cs.ox.ac.uk/projects/concurrency-tools/

(7) «The Theory and Practice of Concurrency» by A. W. Roscoe. Used during NTNU lectures. Prentice Hall 1998.
See http://www.cs.ox.ac.uk/publications/books/concurrency/. PDF of the book exists on the Internet.
Also see http://www.cs.ox.ac.uk/publications/books/concurrency/

(8) XCHANs: Notes on a New Channel Type, by Øyvind Teig, in Communicating Process Architectures 2012 (CPA-2012), Proceedings of the 34th WoTUG Technical Meeting (pages 155-170) P.H. Welch et al. (Eds.) Open Channel Publishing Ltd., 2012 ISBN 978-0-9565409-5-9 © 2012 The authors and Open Channel Publishing Ltd and the authors.
Read paper and presentation at at http://www.teigfam.net/oyvind/pub/pub_details.html#XCHAN

(8) "Concurrent and Real-time Systems: the CSP Approach" by Steve Schneider, 1999. It also treats Timed CSP, not supported in FDR2

(9) «Model checking concurrent RSL with CSPm and FDR2», by Lizeth Tapia and Chris George, May 2008. The United Nations University, UNI-IIST report No. 393

(10) PAT: Process Analysis Toolkit. An Enhanced Simulator, Model Checker and Refinement Checker for Concurrent and Real-time Systems. This also takes CSP, but does not seem to be able to directly import CSPm. Made at Singapore University of Technology and Design; School of Computer Engineering, Nanyang Technological University and School of Computing, National University of Singapore.
Download from http://www.patroot.com.

(11) «The Theory and Practice of Concurrency» by A. W. Roscoe. Used during NTNU lectures. Prentice Hall 1998.
See http://www.cs.ox.ac.uk/publications/books/concurrency/. PDF of the book exists on the Internet.
Also see http://www.cs.ox.ac.uk/publications/books/concurrency/

(12) «The Theory and Practice of Concurrency» by A. W. Roscoe. Used during NTNU lectures. Prentice Hall 1998.
See http:// www.cs.ox.ac.uk/publications/books/concurrency/. PDF of the book exists on the Internet.
Also see http://www.cs.ox.ac.uk/publications/ books/concurrency/