Writing RT-Tester Specifications with CSP

— Communicating Sequential Processes —

Prof. Dr. Jan Peleska
Dr. Cornelia Zahlten

University of Bremen and Verified Systems

14May2000

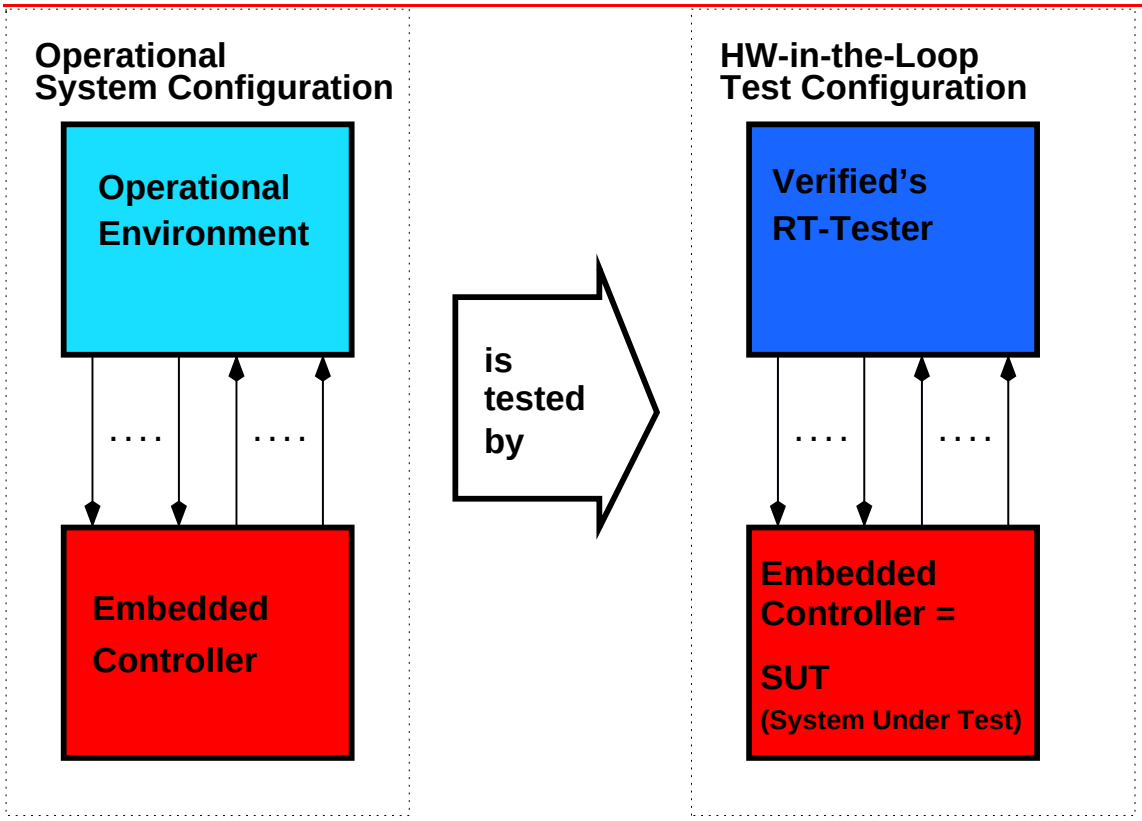
0 Overview

1. **Introduction**
    - **Testing reactive systems**
    - **RT-Tester – basic concepts**
    - **The CSP specification language**
2. Accompanying example: engine controller
3. CSP Test Specification File Structure
4. CSP Data Types
5. CSP Test Specification Interface
6. CSP Processes for writing RT-Tester Specifications

1 Introduction

1.1 Testing Reactive Systems

- **Reactive System:** Computer System continuously interacting with its environment
- **Specification languages:** formalisms to describe networks of timed state machines and variants thereof
- **Test Approach for Reactive Systems:**
  - Simulate components of the operational environment
  - Check whether reactive system responds to inputs as required in the specification


Operational
System Configuration
Operational
Environment
Embedded
Controller
is
tested
by
HW-in-the-Loop
Test Configuration
Verified's
RT-Tester
Embedded
Controller =
SUT
(System Under Test)

RT-Tester performs

- Software integration tests
- Hardware-in-the-loop tests

for reactive systems.

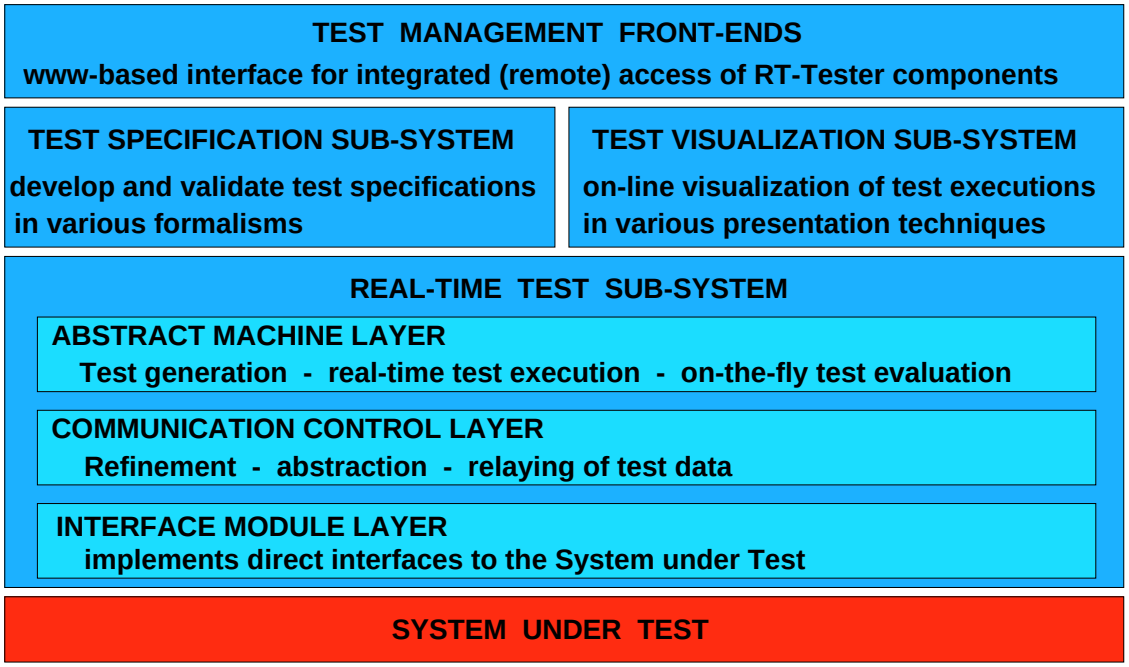
RT-Tester provides automatic

- generation of test data
- test execution in real-time
- on-the-fly test evaluation
- test documentation

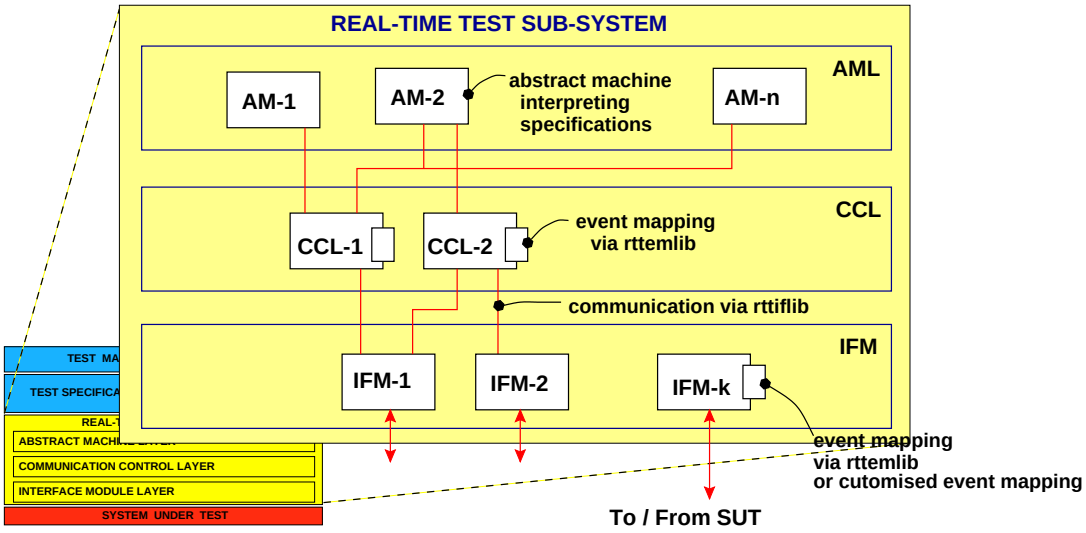
RT-Tester – basic concepts

- RT-Tester **test specifications** are written in CSP, possibly with customised extensions written in C or C++.
- Test specifications are compiled into data structures (called **transition graphs**) which can be interpreted in real-time by **Abstract Machines**.
- In test specifications, events to be passed between Abstract Machines and the system under test (SUT) are described in an abstract way, the mapping to concrete interfaces is performed by **Interface Modules**.
- An RT-Tester **test case** consists of one or more specifications exercised on the SUT in parallel and in real-time.

RT-Tester – basic concepts



RT-Tester – basic concepts



Communicating Sequential Processes (CSP), Hoare 1985

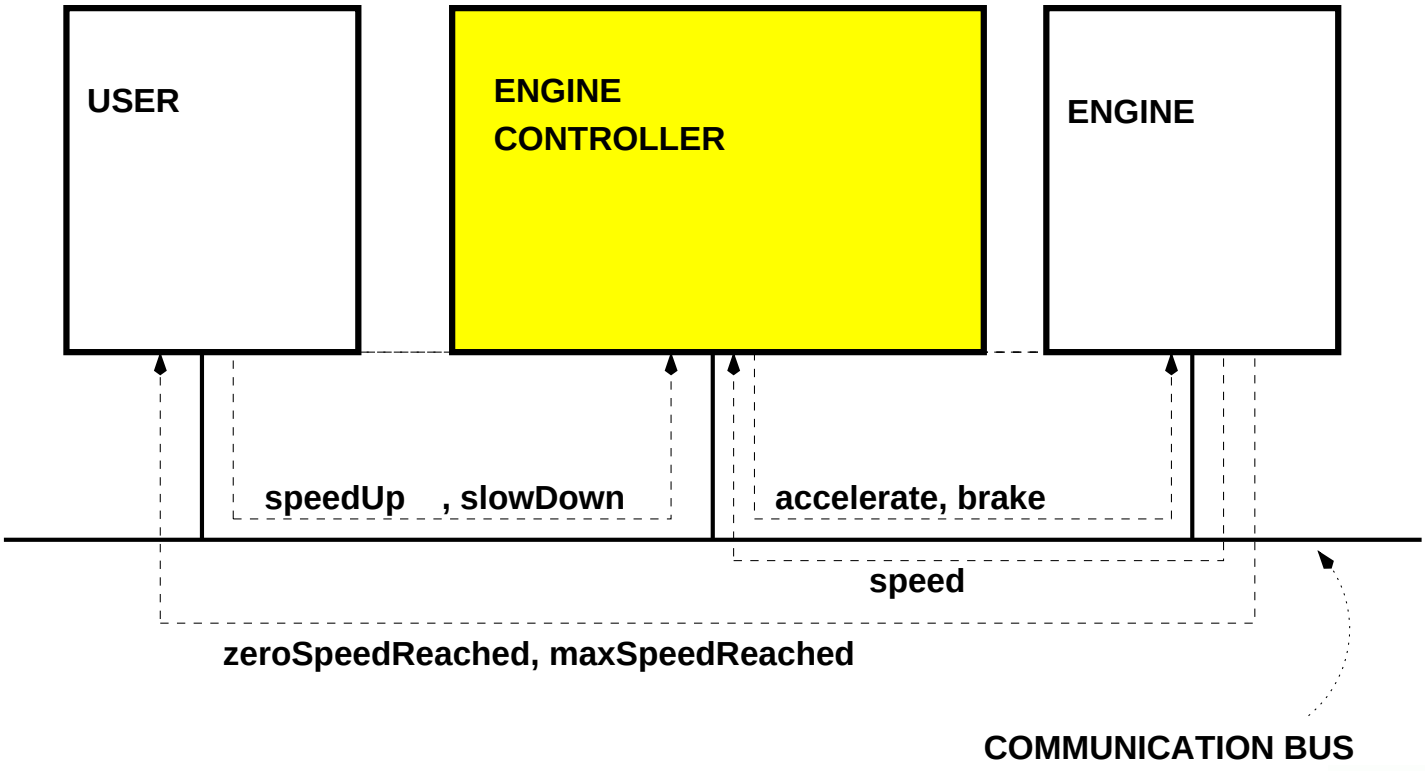
- CSP is a language to describe **networks of processes** which proceed from one state to another by engaging into events.
- A sequential CSP process is a variant of a **timed state machine**.
- Processes may be composed by **operators** and synchronised by **events**.
- Each parallel component must be willing to participate in a given synchronisation event before the whole network can make the transition.
- CSP can be used for **system specification, verification of specifications and specification-based testing**.

1. Introduction
2. **Accompanying example: engine controller**
3. CSP Test Specification File Structure
4. CSP Data Types
5. CSP Test Specification Interface
6. CSP Processes for writing RT-Tester Specifications

"Real" Operational Environment:

- User sends SPEED-UP and SLOW-DOWN commands to controller
- Controller ACCELERATEs and BRAKES the engine according to user command
- Controller triggers EMERGENCY BRAKE after 5sec at top-speed, if user does not give a SLOW-DOWN command before

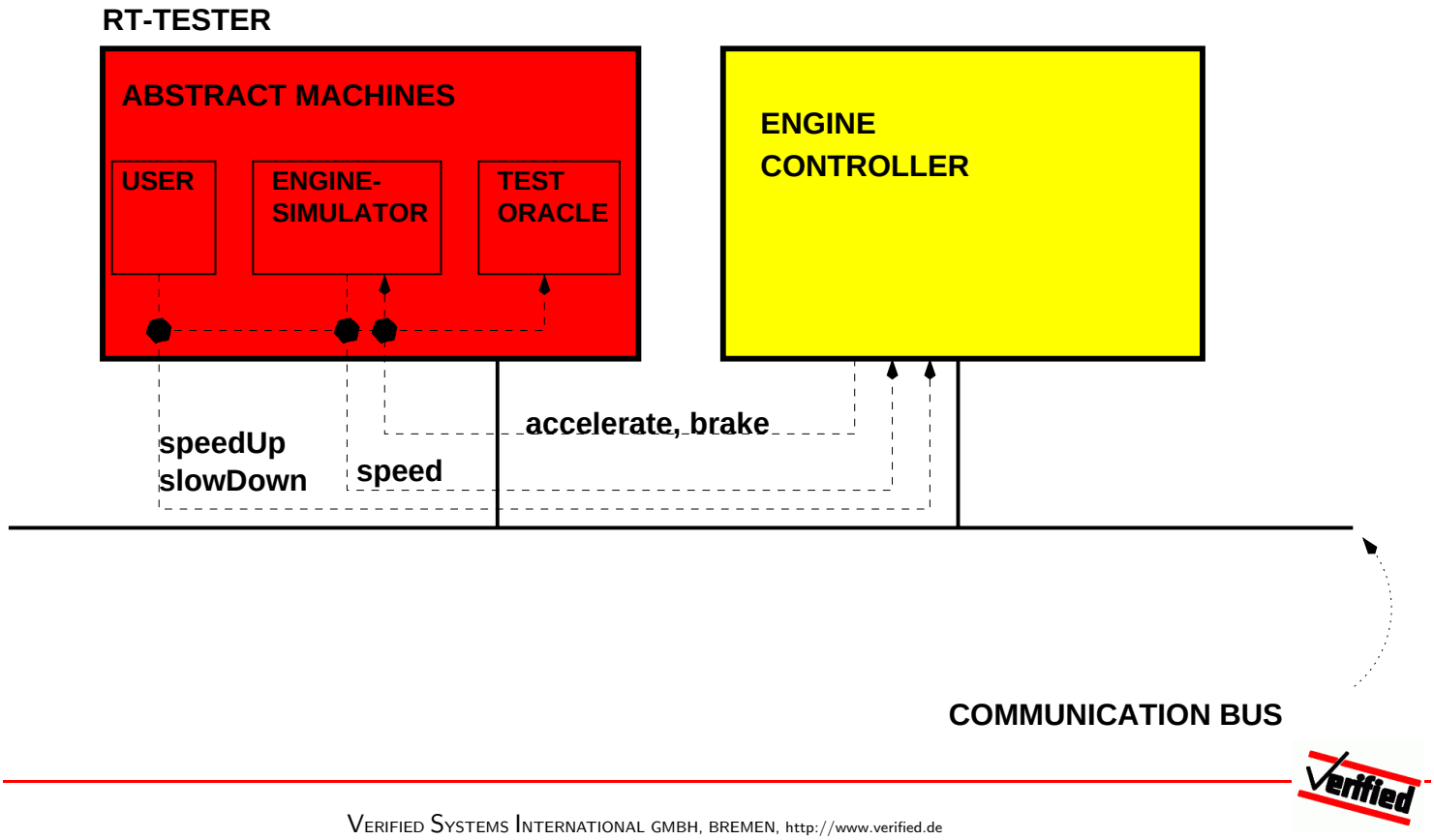
Accompanying example: engine controller



Test Environment:

- RT-Tester Test Engine interfaces to Controller by USER INTERFACE and ENGINE INTERFACE
- RT-Tester Abstract Machine 1 generates all relevant user behaviours
- RT-Tester Abstract Machine 2 simulates engine behaviour
- RT-Tester Abstract Machine 3 acts as TEST ORACLE and automatically checks proper reactions of Controller

Accompanying example: engine controller



1. Introduction
2. Accompanying example: engine controller
3. **CSP Test Specification File Structure**
4. CSP Data Types
5. CSP Test Specification Interface
6. CSP Processes for writing RT-Tester Specifications

3 CSP Test Specification File Structure

CSP specifications are defined by (networks of communicating) sequential CSP processes. Each process definition is written in the form `name = CSP-term`, where `name` is an identifier for a CSP process, and `CSP-term` is a CSP expression containing references to events, processes and the CSP operators to be introduced below.

An RT-Tester CSP specification consists of

1. **data types and constant definitions**
2. **channel declarations** (interface definitions)
3. **included Macro Processes** (pre-defined auxiliary processes)
4. **main process definition** (the test specification process)
5. **subprocess definitions** (top-down decomposition of the main process)

1. Introduction
2. Accompanying example: engine controller
3. CSP Test Specification File Structure
4. **CSP Data Types**
5. CSP Test Specification Interface
6. CSP Processes for writing RT-Tester Specifications

Predefined Types:

`Bool` Boolean values `{true, false}`

`Int` Integer values, e.g. `{0..3} <= Int`

Named Types:

```
DIGIT  = {0,1,2,3,4,5,11,12}
RANGE  = {3..25}
C1     = 17
C2     = off
```

Data Types:

```
datatype BUTTON = pushed | released
datatype Simple_Color = Red | Green | Blue
```

5. **CSP Test Specification Interface**
    - **CSP Channels and Events**
    - **Input/Output Channels**
    - **Error and Warning Channels**
    - **Timer Channels**
    - **Internal Channels**
    - **List of Channel Keywords**
    - **Event Syntax**
6. CSP Processes for writing RT-Tester Specifications

5 CSP Test Specification Interface

5.1 CSP Channels and Events

- In RT-Tester CSP specifications, observable information is transported by **Channels**.
- Channels are classified as
  - **Atomic:** the channel transports an unstructured signal
  - **Structured:** the channel transports structured data items (messages), where each data component is member of a previously defined type.
- An **Event** corresponds to a point in time where
  - an atomic channel signal is triggered, or
  - a data item is passed along structured channels.

5.2 Input/Output Channels

Datatypes:

```
datatype BUTTON = pushed | released
DIGIT  = {0,1,2,3,4,5,11,12}
```

I/O-Channels:

```
--$$AM_OUTPUT
channel Ptt : BUTTON
channel DialDigit: DIGIT.BUTTON
--$$AM_INPUT
channel Called
```

Events:

```
Ptt.pushed
Ptt.released
DialDigit.0.pushed
...
DialDigit.12.released
Called
```

RT-Tester channel declarations are introduced by keywords.

Error Channels:

```
--$$AM_ERROR
channel error : {0..9}
```

If an error event is produced by the specification, RT-Tester produces an error message and stops the test process.

Warning Channels:

```
--$$AM_WARNING
channel warning : {10..19}
```

If a warning event is produced by the specification, RT-Tester produces a warning but the test process goes on.

Timers are special channels to specify and test timing conditions.

Timers:

```
TIMERS  = {1..5}

--$$AM_SET_TIMER
channel setT : TIMERS
--$$AM_ELAPSED_TIMER
channel elaT : TIMERS
--$$AM_RESET_TIMER
channel resT : TIMERS
```

setT: Set a timer to a value of fixed time ticks.

elaT: Timer elapsed. These events are produced by RT-Tester.

resT: Reset a timer to zero, e.g. stop the timer.

Internal channels are used for internal or hidden events.

Internal:
```
--$$AM_INTERNAL
channel myInternalChan : myType
```

Declares channels

- which should occur in the test execution log, but should NOT be sent to the SUT or other AMs (for example: internal events as requirement tags).
- which are hidden by means of the \ - operator.

Keywords:
```
--$$AM_ERROR
--$$AM_WARNING
--$$AM_SET_TIMER
--$$AM_ELAPSED_TIMER
--$$AM_RESET_TIMER
--$$AM_INPUT
--$$AM_OUTPUT
--$$AM_INTERNAL
```

Each keyword introduces a section of channel definitions.
The above order is fixed, e.g. error channels have to be declared first, internal channels have to be declared last.
Keywords may be omitted, if there are no channels of that type.

"pragma" List of Channel Keywords

Keywords:

```
pragma AM_ERROR
pragma AM_WARNING
pragma AM_SET_TIMER
pragma AM_ELAPSED_TIMER
pragma AM_RESET_TIMER
pragma AM_INPUT
pragma AM_OUTPUT
pragma AM_INTERNAL
```

RT-Tester provides two alternatives to introduce keywords: They may start with either `--$$` or `pragma` .
Future versions of RT-Tester will only support the `pragma` version.

```
channel chan :  0..6
channel singlevent
```

`chan!5`	process produces output 5 on channel chan
`chan?5`	process waits for input 5 and accepts only this input
`chan.5`	process produces or accepts only event chan.5 This depends on the keyword AM_INPUT or AM_OUTPUT, which preceeds the channel declaration.
`chan?x`	process waits for any input on this channel
`!singlevent`	same as **`singlevent`**
`?singlevent`	syntax error
`singlevent`	process produces or accepts the event

6. CSP Processes for writing RT-Tester Specifications

- STOP Process
- Events and Prefixing
- If-Then-Else
- Wait-Timers
- Process References
- Recursion
- Parameterised Processes
- Expressions on Parameters
- Sequential Composition
- Macro Processes
- External Choice
- Timeout-Timers
- Parameterised External Choice and Guards
- Internal Choice
- Internal Choice and Timers
- Interrupt Operator
- Hiding
- Interleaving
- Parallel

6 CSP Processes for writing RT-Tester Specifications

6.1 STOP Process

Syntax: `P = STOP`

Description: A process with no action at all.

Test Features: If an input occurs while RT-Tester is testing the process `STOP`, an error message is produced (unexpected input event).

Syntax: `P = a -> Q`

Description: Event 'a' is immediately followed by process Q.

Examples: `P = in?x -> acc!x -> STOP`

Process P waits for an input value on channel `in`, sends that same value to channel `acc`, then stops. `x` is a local process variable, implicitly declared by usage in `in?x`.

Test Features: Any other (non-expected) input is an error.
If a specified output is not produced by P, somebody waits.

6.3 If-Then-Else

Syntax: `X = if (<condition>) then P else Q`

Description: Process `X` behaves as process `P` if `<condition>` evaluates to `true`, otherwise `X` behaves as process `Q`. `<condition>` is a boolean expression over local process variables and process parameters (see below).

Test Features: If-Then-Else may be used to evaluate correctness of SUT responses.

Example:

```
TEST_ENVIRONMENT =
   toSUT!5 -> fromSUT?x ->
           ( if ( x < 5 )
             then (outputOk! -> STOP)
             else (warning! -> STOP ) )
```

Syntax:

```
--$$AM_SET_TIMER
channel setT : {0..9}
--$$AM_ELAPSED_TIMER
channel elaT : {0..9}
 ...
ENV = toSUT!1 -> setT!0 -> elaT.0 -> toSUT!2 -> STOP
```

Description: Wait-timers are realised as pairs of SET-TIMER/ELAPSED-TIMER events. `setT!0` sets timer number 0, and `elaT.0` indicates that the time interval associated with this timer has elapsed. The concrete duration (fixed time or random time interval) is bound to each timer in the RT-Tester configuration file.

Test Features: The test environment uses wait-timers if time has to pass between consecutive outputs sent to the SUT.

Syntax: `P = Q` where `Q` is another defined CSP process.

Description: A process may reference another process by name. For example, this supports the re-use of existing process definitions.

Example:

```
P = fromSUT?x -> ( if ( x == 5 ) then Q else R )

Q = toSUT!0 -> STOP

R = toSUT!1 -> STOP
```

Description: Process references may be applied recursively. This is used to define non-terminating processes. *Ungarded recursion*, that is, unbounded recursive process references without interleaved events, is not allowed (leads to *diverging* processes).

Test Features: Recursion is important to test non-terminating systems: Using recursion, it is possible to generate an arbitrary number of different I/O patterns for communication between environment and SUT.

Example:

```
P = fromSUT?x -> ( if ( x == 5 ) then Q else R )
Q = setT!0 -> elaT.0 -> toSUT!0 -> P
R = toSUT!1 -> Q
```

Syntax: `Q(x1,...,xn) = ...`

Description: A process definition may be parameterised by formal parameters `x1,...,xn`, which are bound to concrete values as soon as the process is referenced by another one, providing the actual parameter values. Formal parameters are implicitly typed.

Example:

```
--$$AM_INPUT
channel fromSUT : {0..9}
--$$AM_OUTPUT
channel toSUT : {0..9}

P = fromSUT?x -> ( if ( x == 5 ) then Q else R(x) )
Q = setT!0 -> elaT.0 -> toSUT!0 -> P
R(z) =  toSUT!(z+1)%10 -> Q
```

Description: In addition to the types introduced above, process parameters may be defined as **sets** and **sequences**.

Set expressions:

```
{}                  : empty set
{ n1..n2 }          : number range set (n1, n2 integers)
{|chan1,chan2,...|}: all events associated with chan-
nels chan1,chan2,...
card(M)             : cardinality of (=number of elements in) set M
empty(M)            : Boolean test for empty set M
member(x,M)         : Boolean test whether x is element of  M
union(M1,M2)        : union of sets M1 and M2
inter(M1,M2)        : intersection of sets M1 and M2
diff(M1,M2)         : set difference M1 \ M2
```

Example:

```
P = Q({})
Q(M) = fromSUT?x -> (if ( member(x,M) or (card(M) > 5) )
                     then (error! -> STOP)
                     else Q(union({x},M)))
```

Expressions on Parameters: Sequences

Sequence expressions:

```
<>              : empty sequence
< n1..n2 >      : number range sequence (n1, n2 integers)
#s              : length of sequence s
null(s)         : Boolean test for empty sequence s
elem(x,s)       : Boolean test whether x is member of sequence s
s1^s2^...       : Concatenation of sequences s1, s2, ...
head(s)         : first element of sequence s
tail(s)         : sequence equal to "s without head(s)"
```

Expressions on Parameters: Local Definitions

Syntax: `let <definition> within <CSP-Term>`

Description: In a CSP term, local definitions – e.g., constants, functions or sets – can be introduced by using a let-within construct in front of the CSP term. In the `<definition>`-part of the let-within construct the same `name = expression` syntax (possibly recursive!) is used as in CSP process definitions.

Example:

```
Q(s) =
  let
    getElem(n,s) = if ( #s < n )
                   then a
                   else ( if ( n == 1 )
                          then head(s)
                          else getElem(n-1,tail(s)) )
  within
  (if ( null(s) ) then STOP else getElem(#s,s) -> Q(tail(s)))
```

Syntax: `P; Q` with defined CSP processes `P` and `Q`

Description: A process `P` may terminate using the `SKIP` termination process. In this situation, another process `Q` may start its activity. This is expressed by the sequential composition operator ";". If `P` does not terminate with `SKIP`, `Q` will never be activated.

Example:

```
X = P;Q
P = in?x -> ( if ( x == 5 ) then SKIP else P )
Q = out!3 -> STOP
```

Description: Parameterised processes may be used to define a library or re-usable CSP macro processes. The RT-Tester macro library is contained in CSP-file `rttmacros.csp`. It can be included using the directive `include "rttmacros.csp"` in the CSP specification file. Note that the directory path is required if the include file is not contained in the working directory.

Example:

```
-- Wait-Timer Macro Process
WAIT(t) = setT!t -> elaT.t -> SKIP

-- process using the WAIT-macro
P = a -> WAIT(0); b -> WAIT(1); c -> P
```

Syntax: `P [] Q` with defined CSP processes `P` and `Q`

Description: `P [] Q` is the process which behaves like `P` or `Q`, depending on whether the initial event is processed by `P` or `Q`, respectively. If both `P` and `Q` may engage into the initial event, the choice between the processes is non-deterministic.

Example:

```
P = (a -> P1) [] (b -> P2)
P1 = ...
P2 = ...
```

Description: External choice can be used to model different process behaviours, depending on whether an expected event occurs in time or a timer elapses.

Example:

```
--$$AM_SET_TIMER
channel  setT : {0}
--$$AM_ELAPSED_TIMER
channel elaT : {0}
--$$AM_OUTPUT
channel toSUT : {0..9}
--$$AM_INPUT
channel fromSUT : {0..9}

P = toSUT!5 -> setT!0 -> Q
Q = elaT.0 -> error! -> STOP
    []
    fromSUT?x -> R(x)
R(x) = ...
```

Syntax: `P(z) = [] x:M @ <bexpr(z,x)>&<chanexpr(z,x)> -> Q(x)` with boolean expression `bexpr(z,x)` in process parameters `z` and local variable `x` and channel expression `<chanexpr(z,x)>` in channels, process parameters `z` and local variable `x` of `P`.

Description: The external choice operator allows the choice to range over events which are specified by means of the parameter expression syntax introduced above: For every `x` in set `M`, a channel event specified by `<chanexpr(z,x)>` may be taken, but only if the boolean guard `<bexpr(z,x)>` evaluates to `true`.

Parameterised External Choice and Guards

Example:

```
-- Process that accepts every event from set M
P = RUN({a,b,c,d})
RUN(M) = [] e:M @ e -> RUN(M)

-- Process that accepts every event from M exactly once
-- and then stops with ready-signal
Q = JUSTONCE({a,b,c,d})
JUSTONCE(M) = ( M == {} ) & ready! -> STOP
              []
              ([] e:M @ e -> M(diff(M,{e})))

-- Process that accepts events from variable number of channels
X = Y({1,2,3,4})
Y(M) = [] i:M @ in.i?x -> PROCESS(i,x);Y(diff(M,{i}))
PROCESS(i,x) = ... -> SKIP
```

Syntax: `P = Q |~| R` with defined CSP processes `Q, R`

Description: Process `P` behaves either like `Q` or like `R`, but the decision is made internally, so that processes cooperating with `P` cannot influence this decision.

Test Features: Internal choice in outputs from the environment to the system under test is exploited by the RT-Tester test generation mechanism to choose outputs according to the test coverage strategy. In mixed input/output internal choice expressions, the test environment may choose to refuse all outputs and wait for an input from SUT instead.

Description: If the test specification contains an internal choice over outputs to SUT and a WAIT-condition, the test environment may choose to wait instead of producing an event.

Test Features: This is variant is helpful if the SUT should behave differently depending on whether it gets a message from the environment in time or not.

Example:

```
ENV = (toSUT!5 -> ENV1)
       |~|
      (WAIT(1);fromSUT?x -> ENV2)
```

6.16 Interrupt Operator

Syntax: `P = Q /\ R` with defined CSP process `Q, R`.

Description: Process `P` will behave like `Q` until it is **interrupted** by an initial event `e` of process `R`. If process `Q` cannot engage into `e` in its present state, `P` will continue behaving like `R` from there on. If `e` is an initial event of `R`, and `Q` in its present state can engage into `e` as well, it is nondeterministic whether `P` will continue to behave as `Q` or "switch" to `R`. In most practical applications, the initial events of `R` are distinct from those `Q` may engage into.

Example:

```
P  = (a -> b -> STOP) /\ (c -> d -> STOP)
behaves identically to
Q = (a -> ((b -> c -> d -> STOP) [] (c -> d -> STOP)))
    [] (c -> d -> STOP)
```

6.17 Hiding

Syntax: `P = Q \ H` with defined CSP process `Q` and set of events `H`.

Description: Applying the hiding operator to `Q` results in a process `P` behaving like `Q`, with all events contained in set `H` no longer observable. Note that hiding over external choices may introduce nondeterminism, because the selected branches are no longer visible.

Hiding

Test Features: If events in a CSP specification should neither occur in the test execution log, nor be communicated between test environment and SUT, these events must be hidden. In practice, this situation occurs if (1) a specification is re-used, but some events should become invisible in the new version or (2) auxiliary events have been introduced to construct a parallel process specification (see below), and the auxiliary events should not be visible.

Example:

```
P = ( a -> b -> STOP) \ {a}
  shows the same behaviour as (b -> STOP).
```

Syntax: `P = Q ||| R` with defined CSP processes `Q, R`

Description: Process `P = Q ||| R` shows the behaviours of `Q` and `R`, running independently and in parallel, without any synchronisation between `Q` and `R`. Note that if `Q` and `R` can produce the same events, it may be impossible to decide from the outside which process is responsible for the event generation.

Interleaving

Test Features: The interleaving operator may be practical if one wishes to disregard certain SUT outputs to the test environment temporarily. If these events are just ignored in the test specification, they will produce an error message. Using a `RUN` process interleaved with the intended specification will avoid these error messages. Observe that with RT-Tester, activities which are always independent from each other should be specified separately, using two Abstract Machines to exercise the corresponding test activities in parallel.

Example:

```
-- process which always accepts events on channels fromSUT1, fromSUT2
-- (Q should only refer to channels different from from-
SUT1, fromSUT2)
P = Q ||| RUN({| fromSUT1, fromSUT2 |})
```

Syntax: `P = Q [| M |] R` with defined CSP processes `Q, R` and set of synchronisation events `M`.

Description: Process `P = Q [| M |] R` shows the behaviour of processes `Q, R` running in parallel and synchronising over all events in set `M`: `Q` may only engage into an event `e` from `M`, if `R` does so at the same time. The event is then produced just once, interpreted as a synchronous communication `e` between both processes. With RT-Tester, events from `M` may be of type INPUT, OUTPUT or INTERNAL. In the first case, the synchronously generated event is also sent to other Abstract Machines and the SUT. In the second case, the event must be sent by the SUT or by other AMs. In the third case, the event is invisible for other AMs or for the SUT.

Test Features: The parallel operator is practical to distribute the different aspects contained in one test specifications onto several parallel CSP processes. For example, a parallel process `Q` may be used to detect a specific communication pattern which triggers a different behaviour in `R`. If the parallelisation of the specification has introduced auxiliary channels bearing no relation to SUT interfaces, it may be useful to hide them.

Example:

```
P = (Q [| {| condition |} |] R(false)) \ {| condition |}

Q = in?x -> (if ( ... )
             then (condition!true -> Q)
             else (condition!false -> Q))

R(c) =
   condition?z -> R(z)
   []
   fromSUT?x -> (if ( c and ... )
                 then R(c)
                 else error! -> R(c))
```