T U M

I N S T I T U T F Ü R I N F O R M A T I K

Focus on processes

Maria Spichkova

TUM-I1115
Juli 11

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-07-I1115-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Focus on processes

Maria Spichkova

July 26, 2011

This paper presents an extension of the formal specification language FOCUS [1] as well as of the methodology *Focus on Isabelle* [4] by the process language and optimization of the FOCUS language to specify some trivial cases implicitly, by the specification semantics. As the starting point of the process language we take a structured, formal model for specification and analysis of work flows [2].

Contents

1 Introduction

Specifying components and system in a formal language is helpful to have a possibility to present also processes within the same language. For these reasons we extend the formal language FOCUS [1] by the theory of processes.

FOCUS has an integrated notion of time and modeling techniques for unbounded networks, provides a number of specification techniques for distributed systems and concepts of refinement. More precisely, we suggest to use one of the versions of the FOCUS language, which was used within "FOCUS on Isabelle".

We present in this paper also an optimization of the FOCUS language to specify some trivial cases implicitly, by the specification semantics.

"FOCUS on Isabelle" [4] is a specification and proof methodology/ framework, where one of the main points during specification phase is an alignment on the future proofs to make them simpler and appropriate for application not only in theory but also in practice. Considering this framework we can influence on the complexity of proofs already doing the specification of systems and their properties, e.g. modifying (reformulating) specification to simplify the proofs[1] for a translated FOCUS specification. Thus, the specification and verification/validation methodologies are treated as a single, joined, methodology with the main focus on the specification part.

Hence, representing processes in FOCUS we can use the advantages of "FOCUS on Isabelle" to prove whether some properties of these process hold.

1.1 Background: Focus

A system in FOCUS is represented by its components that are connected by communication lines called *channels*, and are described in terms of its input/output behavior. The components can interact and also work independently of each other. A specification can be elementary or composite – composite specifications are built hierarchically from the elementary ones.

The channels in this specification framework are *asynchronous communication links* without delays. They are *directed* and generally assumed to be *reliable*, and *order preserving*. Via these channels components exchange information in terms of *messages* of specified types.

In FOCUS any specification characterizes the relation between the *communication histories* for the external *input* and *output channels*. To denote that the (lists of) input and output channel identifiers, I and O, build the syntactic interface of the specification S the notation $(I_S \rhd O_S)$ is used. The formal meaning of a specification is exactly this external *input/output relation*.

The FOCUS specifications can be structured into a number of formulas each characterizing a different kind of property, the most prominent classes of them are *safety* and *liveness properties*. FOCUS supports a variety of *specification styles* which describe system components by logical formulas or by diagrams and tables representing logical formulas.

[1] As the verification system we have chosen Isabelle/HOL [3], an interactive semi-automatic theorem prover for Higher-Order Logic.

The central concept in FOCUS are *streams*, that represent communication histories of *directed channels*. For any set of messages M, M^ω denotes the set of all streams, M^∞ and M^* denote the sets of all infinite and all finite streams respectively, $M^{\underline{\omega}}$ denotes the set of all timed streams, $M^{\underline{\infty}}$ and $M^{\underline{*}}$ denote the sets of all infinite and all finite timed streams respectively. A *timed stream* is represented by a sequence of messages and *time ticks*, the messages are also listed in their order of transmission. The ticks model a discrete notion of time.

The most general style of a FOCUS specification is an A/G style (Assumption/Guarantee style, Assumption/Committment style) – a component is specified in terms of an assumption and a guarantee, what means whenever input from the environment behaves in accordance with the assumption asm, the specified component is required to fulfill the guarantee gar.

Focus operators used in the paper:
An empty stream is represented in FOCUS by $\langle\rangle$.
$\langle x\rangle$ denotes the one element stream consisting of the element x, $\mathsf{ft}.l$ - the first element of the untimed stream (list of elements) l.
$\#s$ denotes the length of the stream s. ith time interval of the stream s is represented by $\mathsf{ti}(s, i)$.
$\mathsf{msg}_n(s)$ denotes a stream s that can have at most n messages at each time interval, and $\mathsf{ts}(r)$ denotes a stream r that has exactly one message at each time interval (so called *time-synchronous* stream).

See [1] and [4] for more background on FOCUS and its extensions.

2 Focus: Optimized Specification

Specifying a component we have often such a case where for some time intervals both conditions hold: local variables still be unchanged and there is no output. This can occur, e.g., if at this time interval the component gets no input or if some preconditions (which are necessary to produce the corresponding nonempty output) don't hold.

In classical FOCUS (as well as in Isabelle/HOL) we need to specify such cases explicitly, otherwise we get an underspecified component that has no information how it must act if it gets no input or if some preconditions don't hold. We suggest to extend the classical FOCUS by possibility of adding a new label *optimized* that extends the specification automatically by so-called $else - case$. Thus, the underspecified cases in the component behavior will be automatically understood as follows: local variables must be unchanged (this implies that system stays in the same state), all output streams at the corresponding time unit must be empty. This optimization doesn't be applicable to component with variables representing timer/countdown, because the variables of this kind must be changed even the component gets no input or even some preconditions, which are necessary to produce the corresponding nonempty output, don't hold.

Having such an optimization we get shorter specifications that are more readable and clear.

The idea of optimization for weak-causal components can be presented as follows. Let *SWeak* be some component with n input channels (streams) $x_1, \ldots, x_n$ of types $MI_1, \ldots, MI_n$ and with m output channels $y_1, \ldots, y_n$ of types $MO_1, \ldots, MO_m$ as well as with k local variables $a_1, \ldots, a_k$ of types $L_1, \ldots, L_k$ respectively, and let the specification looks like one presented below.

In real specifications the formula $j + 1$ is mostly presented for by a number of formulas, because the conjunct

$$\neg \mathit{SomeCondition}_1(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_n, t), a_1, \ldots, a_k) \ \wedge \ \cdots \wedge$$
$$\neg \mathit{SomeCondition}_j(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_n, t), a_1, \ldots, a_k)$$

is too unreadable. These formulas can take in worst case ca. a haft of specification place, and exactly these formulas we can eliminate using the optimized semantics that (implicit) adds them automatically to the specification for all the cases we need to argue about the specification, e.g. if we need to translate it to Isabelle/HOL for verification of some properties.

The optimized specification of the component *SWeak* is also presented below.

For strong causal specification the optimization can be done analogously, but taking into account output delays.

SWeak() — timed

in $x_1 : MI_1, \ldots, x_n : MI_n$

out $y_1 : MO_1, \ldots, y_m : MO_m$

---

local $a_1 \in L_1;\ \ldots;\ a_k \in L_k$

- -

asm

SomeAssumptions

- -

gar

1 $SomeCondition_1(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k) \rightarrow$
$SomeCalculation_1(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k,\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t),a'_1,\ldots,a'_k)$

...

j $SomeCondition_j(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k) \rightarrow$
$SomeCalculation_j(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k,\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t),a'_1,\ldots,a'_k)$

j+1 $\neg SomeCondition_1(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k) \wedge \cdots \wedge$
$\neg SomeCondition_j(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k) \rightarrow$
$a'_1 = a_1 \wedge \ldots \wedge a'_k = a_k \wedge \mathsf{ti}(y_1,t) = \langle\rangle \wedge \ldots \wedge \mathsf{ti}(y_m,t) = \langle\rangle$

SWeakOptim() — timed, optimized

in $x_1 : MI_1, \ldots, x_n : MI_n$

out $y_1 : MO_1, \ldots, y_m : MO_m$

---

local $a_1 \in L_1;\ \ldots;\ a_k \in L_k$

- -

asm

SomeAssumptions

- -

gar

1 $SomeCondition_1(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k) \rightarrow$
$SomeCalculation_1(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k,\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t),a'_1,\ldots,a'_k,)$

...

j $SomeCondition_j(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k) \rightarrow$
$SomeCalculation_j(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),a_1,\ldots,a_k,\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t),a'_1,\ldots,a'_k)$

A number of concrete examples of this optimization is presented in Section 6.

Please note that this kind of optimization is the same for the FOCUS components and the FOCUS processes.

3 Focus: Formal Model of Processes

This paper presents the formal representation in FOCUS of the process language described in [2]. A process is understood there as "an observable activity executed by one or several actors, which might be persons, componets, technical systems, or combinations thereof". Each process has one *entry (activation, start) point* and one *exit (end) point*. An entry point is a special kind of input signal/channel that activates the process, where an exit point is a special kind of output signal/channel that is used to indicate that the process is finished.

According to [2], a process can be defined as an elementary or a composed one, where the composition of any two processes P_1 and P_2 can be sequential $P_1 \ ; \ P_2$ or parallel $P_1 \ || \ P_2$, and for any process P we can define repetitively composed process $P \circlearrowleft_{lpspec}$, where *lpspec* denotes a loop specifier.

Any FOCUS process P (elementary or composed) can be represented by the corresponding FOCUS component specification *PComp*, i.e.

$$[P]^{comp} = PComp$$

We treat a process as a special kind of a FOCUS component that has additionally two channels (one input and one output channel) of special kind. These channels represents the entry and exit points of the process. We specify for any process P its entry and exit points by $Entry(P)$ and $Exit(P)$ respectively.

We suggest to use for a FOCUS process a notation similar to the notation for a FOCUS component (see also [1, 5]). Specifying a process we need to argue about its parameters. For these purposes we use an extended version of the definition from [1] of the semantics of an elementary timed specification to one of an elementary timed *parameterized* specification (see [4]).

Definition:
For any elementary timed parameterized specification S we define its semantics, written $[\![S]\!]$, to be the formula:

$$i_S \in I_S^{\infty} \ \wedge \ p_S \in P_S \ \wedge \ o_S \in O_S^{\infty} \ \wedge \ B_S \tag{1}$$

where i_S and o_S denote lists of input and output channel identifiers, I_S and O_S denote their corresponding types, p_S denotes the list of parameters and P_S denotes their types, B_S is a formula in predicate logic that describes the body of the specification S.

□

The formal correlation between the definition of FOCUS processes and FOCUS components are presented below – separately for elementary and composite processes.

Composite specifications of processes (as well as of components) are built hierarchically from elementary ones using constructors for composition and network description and can be represented in the *graphical*, the *constraint* and *operator* style.

To argue about a mode (active or inactive) of a process P at time interval t we introduce a new predicate *ActiveProcess*:

$$ActiveProcess : ProcessName \times \mathbb{N} \rightarrow \mathbb{B}ool$$

Thus, $ActiveProcess(P, t)$ will denote, that the process P is active during the time interval t.

4 Specification Of An Elementary Process

An elementary process corresponds to an elementary FOCUS specification that has one special input channel of type *Event* (input point of the process that corresponds to an activation signal) as well as one special output of the same type (output point of the process that corresponds to a signal *process is finished*). Thus, we don't need to specify the type of this two channels explicitly, because it is given by the purposed syntax.

process ProcessName(Parameters) FrameLabel

entry *StartChannel*

exit *StopChannel*

InputOutputChannels

local *Local_Declarations*
orac *Oracle_Declarations*
univ *Logical_Declarations*

init *Initial_Values_Requirements*

initProcess *Initial_Values_Requirements_For_Every_Process_Restart*

asm
Assumptions

gar
Guaranties

Any process specification can be represented by a FOCUS component using the following translation:

- Each input channel (except the activation signals channel) c has a corresponding buffer (local variable) *cBuffer* of size one (one element buffer), which value will be taking into account, when starting the process.

- The component gets a local variable *active* of type $\mathbb{B}ool$ to represent whether the process is in active phase.

- If the process is inactive, there is no values on its output channels.

The initProcess process specification section differs from a standard FOCUS specification section init in the following sense: everything that is defined within the init section must be initialized only ones, in the beginning, but everything that is defined within the initProcess section must be initialized every time the process is (re)started, i.e. every time the value of the local variable *active* is triggered fom false to true.

4.1 Representation of universal and special parts of a process

Let a process has n input channels $x_1, \ldots, x_n$ and m outputs $y_1, \ldots, y_m$, the condition of its finishing is defined by the predicate *EndingCondition* over the received input values, i.e. over the values saved in the local variables $x_1Buffer, \ldots, x_nBuffer$. By the predicate *Calculations* we represent here all the calculations over the input values that are performed during the process. In some cases we need to extend this predicate by calculations of some other local variables of the process.

All inactive processes behave in a similar manner. An universal part of behavior for a weakly-causal process can be represented as follows:

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(ent, t) = \langle\rangle \ \rightarrow$$
$$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge \ active' = active \ \wedge \ \mathsf{ti}(y_1, t) = \langle\rangle \ \wedge \ \ldots \ \wedge \ \mathsf{ti}(y_m, t) = \langle\rangle$$

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(ent, t) \neq \langle\rangle \ \rightarrow$$
$$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge \ active' = \mathsf{true} \ \wedge \ \mathsf{ti}(y_1, t) = \langle\rangle \ \wedge \ \ldots \ \wedge \ \mathsf{ti}(y_m, t) = \langle\rangle \ \wedge$$
$$Initial\_Values\_Requirements\_For\_Every\_Process\_Restart$$

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_1, t) \neq \langle\rangle \ \rightarrow \ x_1Buffer' = \mathsf{ft.ti}(x_1, t)$$
$$\ldots$$
$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_n, t) \neq \langle\rangle \ \rightarrow \ x_nBuffer' = \mathsf{ft.ti}(x_n, t)$$

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_1, t) = \langle\rangle \ \rightarrow \ x_1Buffer' = x_1Buffer$$
$$\ldots$$
$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_1, t) = \langle\rangle \ \rightarrow \ x_nBuffer' = x_nBuffer$$

An universal part of behavior for a strongly-causal process is the same modulo delay of one time unit (in special cases, of a number of time units):

$$\mathsf{ti}(ext, 0) = \langle\rangle \ \wedge \ \mathsf{ti}(y_1, 0) = \langle\rangle \ \wedge \ \ldots \ \wedge \ \mathsf{ti}(y_m, 0) = \langle\rangle \qquad (ZeroTimeUnit)$$

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(ent, t) = \langle\rangle \ \rightarrow$$
$$\mathsf{ti}(ext, t+1) = \langle\rangle \ \wedge \ active' = active \ \wedge \ \mathsf{ti}(y_1, t+1) = \langle\rangle \ \wedge \ \ldots \ \wedge \ \mathsf{ti}(y_m, t+1) = \langle\rangle$$

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(ent, t) \neq \langle\rangle \ \rightarrow$$
$$\mathsf{ti}(ext, t+1) = \langle\rangle \ \wedge \ active' = \mathsf{true} \ \wedge \ \mathsf{ti}(y_1, t+1) = \langle\rangle \ \wedge \ \ldots \ \wedge \ \mathsf{ti}(y_m, t+1) = \langle\rangle \ \wedge$$
$$Initial\_Values\_Requirements\_For\_Every\_Process\_Restart$$

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_1, t) \neq \langle\rangle \ \rightarrow \ x_1 Buffer' = \mathsf{ft.ti}(x_1, t)$$
$$\ldots$$
$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_n, t) \neq \langle\rangle \ \rightarrow \ x_n Buffer' = \mathsf{ft.ti}(x_n, t)$$

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_1, t) = \langle\rangle \ \rightarrow \ x_1 Buffer' = x_1 Buffer$$
$$\ldots$$
$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(x_1, t) = \langle\rangle \ \rightarrow \ x_n Buffer' = x_n Buffer$$

According to purposed syntax the formulas presented above can be omitted within a FOCUS process specification (except the formula we called *ZeroTimeUnit* that belongs only to strongly-causal specifications), because they are specified implicitly by using this kind of FOCUS specifications.

The *Initial_Values_Requirements_For_Every_Process_Restart* formula will be moved to the corresponding specification section.

The next two formulas are also similar for every process:

$$active = \mathsf{true} \ \wedge \ EndingCondition(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_n, t), x_1 Buffer, \ldots, x_n Buffer) \ \rightarrow$$
$$CalculationsF(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_n, t), x_1 Buffer, \ldots, x_n Buffer,$$
$$x_1 Buffer', \ldots, x_n Buffer', \mathsf{ti}(y_1, t), \ldots, \mathsf{ti}(y_m, t)) \wedge$$
$$\mathsf{ti}(ext, t) = \langle event \rangle \ \wedge \ active' = \mathsf{false}$$

$$active = \mathsf{true} \ \wedge \ \neg EndingCondition(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_n, t), x_1 Buffer, \ldots, x_n Buffer) \ \rightarrow$$
$$Calculations(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_n, t), x_1 Buffer, \ldots, x_n Buffer,$$
$$x_1 Buffer', \ldots, x_n Buffer', \mathsf{ti}(y_1, t), \ldots, \mathsf{ti}(y_m, t)) \wedge$$
$$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge \ active' = active$$

The predicate *EndingCondition* is uses to specify the ending condition of the process – if its value is true, then the process is finished.

The predicate *Calculations* describes the calculations of the output values for the current step (for the current time unit) and of the buffer values for the next step (for the next time unit).

The predicate *CalculationsF* describes the calculations of the output and buffer values for a special case when the *EndingCondition* holds – for this case we often need some simpler kind of calculation than specified within *Calculations* or, even, of some other kind, but sometimes we can use a single predicate for both

cases.

We purpose to represent these formulas within a FOCUS process specification using a simplified syntax (omitting the conjunct $active = \mathsf{true}$):

$$
\begin{aligned}
&\mathit{EndingCondition}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer}) \;\rightarrow\\
&\mathit{CalculationsF}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer},\\
&\qquad\qquad x_1\mathit{Buffer}',\ldots,x_n\mathit{Buffer}',\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t)) \;\wedge\\
&\mathsf{ti}(\mathit{ext},t) = \langle \mathit{event}\rangle \;\wedge\; \mathit{active}' = \mathsf{false}
\end{aligned}
$$

$$
\begin{aligned}
&\neg\mathit{EndingCondition}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer}) \;\rightarrow\\
&\mathit{Calculations}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer},\\
&\qquad\qquad x_1\mathit{Buffer}',\ldots,x_n\mathit{Buffer}',\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t)) \;\wedge\\
&\mathsf{ti}(\mathit{ext},t) = \langle\rangle \;\wedge\; \mathit{active}' = \mathit{active}
\end{aligned}
$$

Thus, a general FOCUS process specification *Process* looks like follows[2]:

process P() — timed

entry *start*

exit *stop*

---

in $x_1 : MI_1, \ldots, x_n : MI_n$

out $y_1 : MO_1, \ldots, y_m : MO_m$

---

local $\mathit{active} : \mathbb{B}\mathrm{ool};\ x_1\mathit{Buffer} \in MI_1;\ \ldots;\ x_n\mathit{Buffer} \in MI_n$

---

init $\mathit{active} = \mathsf{false};$
$x_1\mathit{Buffer} = \mathit{BufferInitValue}_1;\ \ldots;\ x_n\mathit{Buffer} = \mathit{BufferInitValue}_n$

---

initProcess *Initial_Values_Requirements_For_Every_Process_Restart*

---

asm
SomeAssumptions

---

gar

1 $\mathit{PEndingCondition}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer}) \;\rightarrow$
$\mathit{PCalculationsF}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer},$
$x_1\mathit{Buffer}',\ldots,x_n\mathit{Buffer}',\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t)) \;\wedge$
$\mathsf{ti}(\mathit{ext},t) = \langle \mathit{event}\rangle \;\wedge\; \mathit{active}' = \mathsf{false}$

2 $\neg\mathit{PEndingCondition}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer}) \;\rightarrow$
$\mathit{PCalculations}(\mathsf{ti}(x_1,t),\ldots,\mathsf{ti}(x_n,t),x_1\mathit{Buffer},\ldots,x_n\mathit{Buffer},$
$x_1\mathit{Buffer}',\ldots,x_n\mathit{Buffer}',\mathsf{ti}(y_1,t),\ldots,\mathsf{ti}(y_m,t)) \;\wedge$
$\mathsf{ti}(\mathit{ext},t) = \langle\rangle \;\wedge\; \mathit{active}' = \mathsf{true}$

[2]We are focusing here on timed specifications. That implies that we always use a frame label *timed*.

$MI_1, \ldots, MI_n$ and $MO_1, \ldots, MO_m$ are here the data types of input and output streams respectively, and
$BufferInitValue_1, \ldots, BufferInitValue_n$ are the initial values of buffers for the input channels $x_1, \ldots, x_n$.

A process specification can have in general a number of other local variables as well as parameters, we omit this here to concentrate on the main idea of the general representation.

On the place of *SomeAssumptions* all the assumption formulas that are needed for the process specification must be defined.

The predicates *PEndingCondition* and *PNecessaryCalculations* must be defined extra (*P* states here for the name of the process).

Please note, that we speak here about weakly-causal specifications. Specifying a strongly-causal process we need to add to the specification the *ZeroTimeUnit* part and take care about the delays by the output channels. A number of examples of strongly-causal specifications are given in Section 6 discussing a case study *Pumping Station*.

Please also note: if a specification has some other local variables, they can also be parameters of the *EndingCondition* and/or *Calculations* predicates. Contrariwise, these predicates not always need to have all of the parameters that are presented here, e.g., the *EndingCondition* may have also as parameters merely $\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_n, t)$ or merely $x_1 Buffer, \ldots, x_n Buffer$.

If a process P will be specified by TSTD (timed state transition diagram), then the *Local_Declarations* field must contain a local (state) variable with the corresponding name *PSt*.

In the case the process behavior assumes that the TSTD must be activated every time at some initial state $State_0$, than we need to add to the *Initial_Values_Requirements_For_Every_Process_Restart* the corresponding definition: $PSt = S_0$.

Doing the translation of the FOCUS process specification to the corresponding FOCUS component specification, we remove the **initProcess** section with the formula $PSt = S_0$ and extend the formula

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(ent, t) \neq \langle\rangle \ \rightarrow$$
$$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge \ active' = \mathsf{true} \ \wedge \ \mathsf{ti}(y_1, t) = \langle\rangle \ \wedge \ \ldots \ \wedge \ \mathsf{ti}(y_m, t) = \langle\rangle$$

as follows (important: here we define the value of *PSt* not for the current time unit, but for the next time unit that will be the first time unit after (re)start):

$$active = \mathsf{false} \ \wedge \ \mathsf{ti}(ent, t) \neq \langle\rangle \ \rightarrow$$
$$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge \ active' = \mathsf{true} \ \wedge \ \mathsf{ti}(y_1, t) = \langle\rangle \ \wedge \ \ldots \ \wedge \ \mathsf{ti}(y_m, t) = \langle\rangle \ \wedge \ PSt' = S_0$$

4.2 Representation of process by component

This FOCUS process specification *Process* corresponds to the following FOCUS component specification *ProcessComp*:

ProcessComp() — timed

in $start : Event;\ x_1 : MI_1, \ldots, x_n : MI_n$

out $stop : Event;\ y_1 : MO_1, \ldots, y_m : MO_m$

---

local $active : \mathbb{B}ool;\ x_1Buffer \in MI_1;\ \ldots;\ x_nBuffer \in MI_n$

- -

init $active = \mathsf{false};$
$x_1Buffer = BufferInitValue_1;\ \ldots;\ x_nBuffer = BufferInitValue_n$

- -

asm
SomeAssumptions

- -

gar

1 $active = \mathsf{false} \ \wedge\ \mathsf{ti}(ent, t) = \langle\rangle \ \rightarrow$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = active \ \wedge\ \mathsf{ti}(y_1, t) = \langle\rangle \ \wedge\ \ldots \ \wedge\ \mathsf{ti}(y_m, t) = \langle\rangle$

2 $active = \mathsf{false} \ \wedge\ \mathsf{ti}(ent, t) \neq \langle\rangle \ \rightarrow$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = \mathsf{true} \ \wedge\ \mathsf{ti}(y_1, t) = \langle\rangle \ \wedge\ \ldots \ \wedge\ \mathsf{ti}(y_m, t) = \langle\rangle$

3 $active = \mathsf{false} \ \wedge\ \mathsf{ti}(x_1, t) \neq \langle\rangle \ \rightarrow\ x_1Buffer' = \mathsf{ft.ti}(x_1, t)$

...

3+n $active = \mathsf{false} \ \wedge\ \mathsf{ti}(x_n, t) \neq \langle\rangle \ \rightarrow\ x_nBuffer' = \mathsf{ft.ti}(x_n, t)$

4+n $active = \mathsf{false} \ \wedge\ \mathsf{ti}(x_1, t) = \langle\rangle \ \rightarrow\ x_1Buffer' = x_1Buffer$

...

4+2n $active = \mathsf{false} \ \wedge\ \mathsf{ti}(x_1, t) = \langle\rangle \ \rightarrow\ x_nBuffer' = x_nBuffer$

5+2n $active = \mathsf{true} \ \wedge$
$PEndingCondition(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_1, t), x_1Buffer, \ldots, x_nBuffer) \ \rightarrow$
$PCalculationsF(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_1, t), x_1Buffer, \ldots, x_nBuffer,$
$x_1Buffer', \ldots, x_nBuffer', \mathsf{ti}(y_1, t), \ldots, \mathsf{ti}(y_m, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle event \rangle \ \wedge\ active' = \mathsf{false} \ \wedge$

6+2n $active = \mathsf{true} \ \wedge$
$\neg PEndingCondition(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_1, t), x_1Buffer, \ldots, x_nBuffer) \ \rightarrow$
$PCalculations(\mathsf{ti}(x_1, t), \ldots, \mathsf{ti}(x_1, t), x_1Buffer, \ldots, x_nBuffer,$
$x_1Buffer', \ldots, x_nBuffer', \mathsf{ti}(y_1, t), \ldots, \mathsf{ti}(y_m, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = \mathsf{true}$

4.3 Example: Specification of an elementary process

To give an example, let us discuss a trivial FOCUS specification of the process *NumProc* that outputs via an output channel *evens* a sequence of even numbers, which are smaller than the last natural number received by the process via an input stream *number*. E.g., if the last received number was 9, then after the activation *entry*-signal the process outputs the following sequence of numbers: 8, 6, 4, 2, 0 (and after that indicates by the *exit*-signal that the process is finished).

The predicate *NumEndingCondition* is defined here by $numberBuffer \leq 1$ and its negation can be simply represented by $numberBuffer > 1$:

NumEndingCondition

$b \in \mathbb{N}$

$b \leq 1$

The predicates *NumCalculations* and *NumCalculationsF* can be specified for this example as follows[3]:

NumCalculations

$x, xNext \in \mathbb{N};\ y \in \mathbb{N}^*$

$\mathsf{even}(x) \ \rightarrow \ xNext = x - 2 \ \wedge \ y = \langle x \rangle$

$\neg\mathsf{even}(x) \ \rightarrow \ xNext = x - 3 \ \wedge \ y = \langle x - 1 \rangle$

NumCalculationsF

$x, xNext \in \mathbb{N};\ y \in \mathbb{N}^*$

$\mathsf{even}(x) \ \rightarrow \ xNext = 0 \ \wedge \ y = \langle x \rangle$

$\neg\mathsf{even}(x) \ \rightarrow \ xNext = 0 \ \wedge \ y = \langle x - 1 \rangle$

Because the *NumEndingCondition* predicate is here very simple, we can simlify the specification vs. the general representation: cf. the two versions below.

[3] If we use here the predicate *NumCalculations* also for the case the predicate *NumEndingCondition* holds, we get the equalities like $xNext = 0 - 2$ and $xNext = 1 - 3$ that is not fully correct for natural numbers in Focus.
If we presume for Focus also the Isabelle/HOL rule like $0 - x = 0$, we can use the predicate *NumCalculations* for both cases, but this solution is less intuitive and not very clean.

process $\mathrm{Num}()$ — timed

entry ent

exit ext

---

in $number : \mathbb{N}$

out $evens : \mathbb{N}$

---

local $active : \mathbb{B}\mathrm{ool}$; $numberBuffer : \mathbb{N}$

---

init $active = \mathsf{false}$; $numberBuffer = 0$

---

asm $\mathsf{msg}_1(number)$

---

gar

$\forall\, t \in \mathbb{N}$:

1 $active = \mathsf{true} \ \wedge \ NumEndingCondition(numberBuffer) \ \rightarrow$
$NumCalculationsF(numberBuffer, numberBuffer', \mathsf{ti}(evens, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle event \rangle \ \wedge \ active' = \mathsf{false}$

2 $active = \mathsf{true} \ \wedge \ \neg NumEndingCondition(numberBuffer) \ \rightarrow$
$NumCalculations(numberBuffer, numberBuffer', \mathsf{ti}(evens, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge \ active' = active$

process $\mathrm{Num}()$ — timed

entry ent

exit ext

---

in $number : \mathbb{N}$

out $evens : \mathbb{N}$

---

local $numberBuffer : \mathbb{N}$; $active : \mathbb{B}\mathrm{ool}$

---

init $numberBuffer = 0$; $active = \mathsf{false}$

---

asm $\mathsf{msg}_1(number)$

---

gar

$\forall\, t \in \mathbb{N}$:

1 $active = \mathsf{true} \ \wedge \ numberBuffer \leq 1 \ \rightarrow$
$NumCalculationsF(numberBuffer, numberBuffer', \mathsf{ti}(evens, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle event \rangle \ \wedge \ active' = \mathsf{false}$

2 $active = \mathsf{true} \ \wedge \ numberBuffer > 1 \ \rightarrow$
$NumCalculations(numberBuffer, numberBuffer', \mathsf{ti}(evens, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge \ active' = active$

Please note that specifying this process we don't need the initProcess section, because we don't need to set some variables to some standard values every time the process is (re)started.

This process can be directly presented by the corresponding specification of a FOCUS component *NumComponet* as follows.

NumComponent() timed

in $ent : Event;\ number : \mathbb{N}$

out $ext : Event;\ evens : \mathbb{N}$

local $active : \mathbb{B}ool;\ numberBuffer : \mathbb{N}$

init $active = \mathsf{false};\ numberBuffer = 0$

asm $\mathsf{ts}(number)$

gar

$\forall\, t \in \mathbb{N}$:

1 $active = \mathsf{false} \ \wedge\ \mathsf{ti}(ent, t) = \langle\rangle \ \rightarrow$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = active \ \wedge\ \mathsf{ti}(evens, t) = \langle\rangle$

2 $active = \mathsf{false} \ \wedge\ \mathsf{ti}(ent, t) \neq \langle\rangle \ \rightarrow$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = \mathsf{true} \ \wedge\ \mathsf{ti}(evens, t) = \langle\rangle$

3 $active = \mathsf{false} \ \wedge\ \mathsf{ti}(number, t) \neq \langle\rangle \ \rightarrow\ numberBuffer' = \mathsf{ft}.\mathsf{ti}(number, t)$

4 $active = \mathsf{false} \ \wedge\ \mathsf{ti}(number, t) = \langle\rangle \ \rightarrow\ numberBuffer' = numberBuffer$

5 $active = \mathsf{true} \ \wedge\ numberBuffer \leq 1 \ \rightarrow$
$NumCalculationsF(numberBuffer, numberBuffer', \mathsf{ti}(evens, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle event \rangle \ \wedge\ active' = \mathsf{false}$

6 $active = \mathsf{true} \ \wedge\ numberBuffer > 1 \ \rightarrow$
$NumCalculations(numberBuffer, numberBuffer', \mathsf{ti}(evens, t)) \ \wedge$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = active$

The sixth formula of the specification above can be also simplified (vs. the using of auxiliary predicates) into two formulas as follows:

$active = \mathsf{true} \ \wedge\ numberBuffer > 1 \ \wedge\ \mathsf{even}(numberBuffer) \ \rightarrow$
$numberBuffer' = numberBuffer - 2 \ \wedge$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = \mathsf{true} \ \wedge\ \mathsf{ti}(evens, t) = \langle numberBuffer \rangle$

$active = \mathsf{true} \ \wedge\ numberBuffer > 1 \ \wedge\ \neg\mathsf{even}(numberBuffer) \ \rightarrow$
$numberBuffer' = numberBuffer - 3 \ \wedge$
$\mathsf{ti}(ext, t) = \langle\rangle \ \wedge\ active' = \mathsf{true} \ \wedge\ \mathsf{ti}(evens, t) = \langle numberBuffer - 1 \rangle$

5 Specification of Composed Processes P and Q

Let P and Q be any two processes. The sets of input and output channels are defined for processes P and Q as well as for the the corresponding components $PComp$ and $QComp$, where $PComp = [P]^{comp}$ and $QComp = [Q]^{comp}$, as follows:

$$\begin{array}{ll} Entry(P) = entP & Entry(Q) = entQ \\ Exit(P) = extP & Exit(Q) = extQ \\ I_P = i_1, \ldots, i_m & I_Q = x_1, \ldots, x_k \\ O_P = o_1, \ldots, o_n & O_Q = y_1, \ldots, y_z \end{array}$$

$$\begin{array}{ll} I_{[P]^{comp}} = entP, i_1, \ldots, i_m & I_{[Q]^{comp}} = entQ, x_1, \ldots, x_k \\ O_{[P]^{comp}} = extP, o_1, \ldots, o_n & O_{[Q]^{comp}} = extQ, y_1, \ldots, y_z \end{array}$$

5.1 Specification Of An Sequentially Composed Process $P;\ Q$

$$[P(i_1, \ldots, i_m, o_1, \ldots, o_n);\ Q(x_1, \ldots, x_k, y_1, \ldots, y_z)]^{comp} = P^{spec}(entP, i_1, \ldots, i_m, extP, o_1, \ldots, o_n) \otimes Q^{spec}(extP, x_1, \ldots, x_k, extQ, y_1, \ldots, y_z)$$

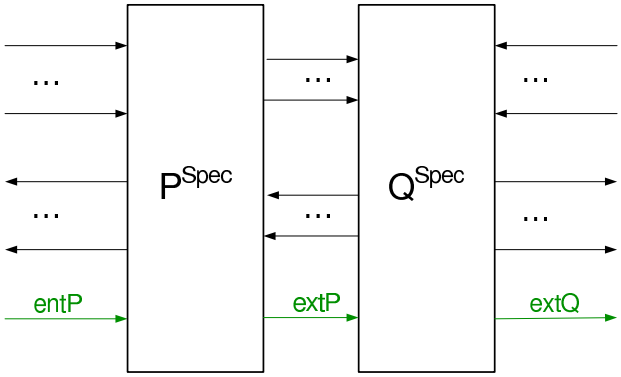


Figure 1: Sequentially Composed Process $P;\ Q$

Fig. 1 shows a graphical representation of sequential composition of two processes P and Q in general. Please note that we draw here in green all the channels, which represent entry and exit points of a process, a well as auxiliary component to merge (in later examples also: to split) the streams over these channels. We draw in blue all the auxiliary components to merge regular channels according to the process composition as well as the corresponding channels.

The sets of input, output and local channels are defined for processes and components as follows:

$$Entry(Q) = entQ = extP = ExitP$$
$$Entry(P;\ Q) = entP$$
$$Exit(P;\ Q) = extQ$$
$$I_{P;\,Q} = (I_P \setminus L_{P;\,Q}) \cup (I_Q \setminus L_{P;\,Q})$$
$$O_{P;\,Q} = (O_P \setminus L_{P;\,Q}) \cup (O_Q \setminus L_{P;\,Q})$$
$$L_{P;\,Q} = (I_P \cap O_Q) \cup (O_P \cap I_Q)$$

$$I_{[P;\,Q]^{comp}} = (I_{[P]^{comp}} \setminus L_{[P;\,Q]^{comp}}) \cup (I_{[Q]^{comp}} \setminus L_{[P;\,Q]^{comp}})$$

$$O_{[P;\,Q]^{comp}} = (O_{[P]^{comp}} \setminus L_{[P;\,Q]^{comp}}) \cup (O_{[Q]^{comp}} \setminus L_{[P;\,Q]^{comp}})$$

$$L_{[P;\,Q]^{comp}} = (I_{[P]^{comp}} \cap O_{[Q]^{comp}}) \cup (O_{[P]^{comp}} \cap I_{[Q]^{comp}})$$

5.2 Specification Of An Alternatively Process $P \oplus Q$

$$[P(i_1,\ldots,i_m,o_1,\ldots,o_n) \oplus Q(x_1,\ldots,x_k,y_1,\ldots,y_z)]^{comp} =$$
$$P^{spec}(entP, i_1,\ldots,i_m, extP, o_1,\ldots,o_n)\otimes$$
$$Q^{spec}(entQ, x_1,\ldots,x_k, extQ, y_1,\ldots,y_z)\otimes$$
$$GuardEvent(entPQ, entP, entQ)\otimes$$
$$Merge(Event)(extP, extQ, extPQ)$$

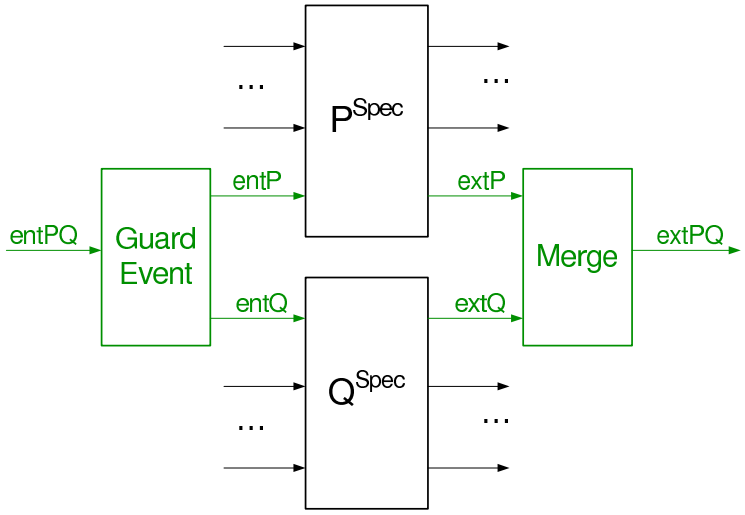


Figure 2: Alternatively Composed Process $P \oplus Q$

Fig. 2 shows a graphical representation of alternative composition of two processes P and Q in general. The special components *GuardEvent* and *Merge* are specified below.

The component *GuardEvent* can be also defined in another way, if the input control flow is represented, e.g., by a stream of natural numbers or if this component has not a single input control flow, but a number of them.

GuardEvent() — timed

in $z : \mathbb{B}\text{ool}$

out $x, y : Event$

asm

$\mathsf{ts}(z)$

gar

$\forall\, t \in \mathbb{N}:$

$\mathsf{ti}(z,t) = \langle\rangle \;\rightarrow\; \mathsf{ti}(x,t) = \langle\rangle \wedge \mathsf{ti}(y,t) = \langle\rangle$

$\mathsf{ti}(z,t) = \langle \mathsf{true} \rangle \;\rightarrow\; \mathsf{ti}(x,t) = \langle event \rangle \wedge \mathsf{ti}(y,t) = \langle\rangle$

$\mathsf{ti}(z,t) = \langle \mathsf{false} \rangle \;\rightarrow\; \mathsf{ti}(x,t) = \langle\rangle \wedge \mathsf{ti}(y,t) = \langle event \rangle$

Merge(type M) — timed

in $x, y : M;$

out $z : M$

asm

$\mathsf{msg}_1(x) \;\wedge\; \mathsf{msg}_1(y)$

gar

$\forall\, t \in \mathbb{N}:$

$\mathsf{ti}(z,t) = \mathsf{ti}(x,t) \frown \mathsf{ti}(y,t)$

The sets of input, output and local channels are defined for processes and components as follows:

$Entry(P \oplus Q) = entPQ$

$Exit(P \oplus Q) = extPQ$

$I_{P \oplus Q} = I_P \cup I_Q$

$O_{P \oplus Q} = O_P \cup O_Q$

$L_{P \oplus Q} = \varnothing$

The sets of input, output and local channels are defined for processes and components as follows:

$$
\begin{aligned}
I_{[P \oplus Q]^{comp}} &= ((I_{[P]^{comp}} \cup I_{[Q]^{comp}}) \setminus \{entP, entQ\}) \cup \{entPQ\} \\
O_{[P \oplus Q]^{comp}} &= ((O_{[P]^{comp}} \cup O_{[Q]^{comp}}) \setminus \{extP, extQ\}) \cup \{extPQ\} \\
L_{[P \oplus Q]^{comp}} &= \{entP, entQ, extP, extQ\}
\end{aligned}
$$

5.3 Specification Of A Simultaneously Composed Process $P \parallel Q$

$$
\begin{aligned}
&[P(i_1,\ldots,i_m,o_1,\ldots,o_n) \parallel Q(x_1,\ldots,x_k,y_1,\ldots,y_z)]^{comp} \;=\\
&\quad P^{spec}(entP, i_1,\ldots,i_m, extP, o_1,\ldots,o_n)\otimes\\
&\quad Q^{spec}(entP, x_1,\ldots,x_k, extQ, y_1,\ldots,y_z)\otimes\\
&\quad ConjParEvent(extP, extQ, extPQ)
\end{aligned}
$$

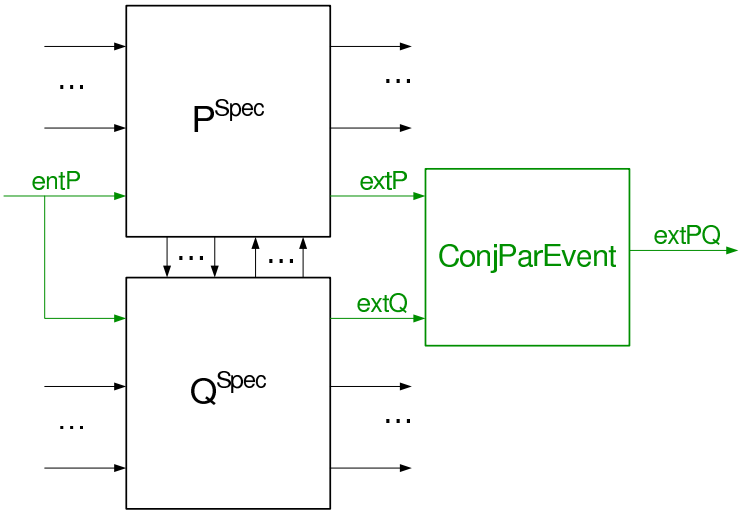


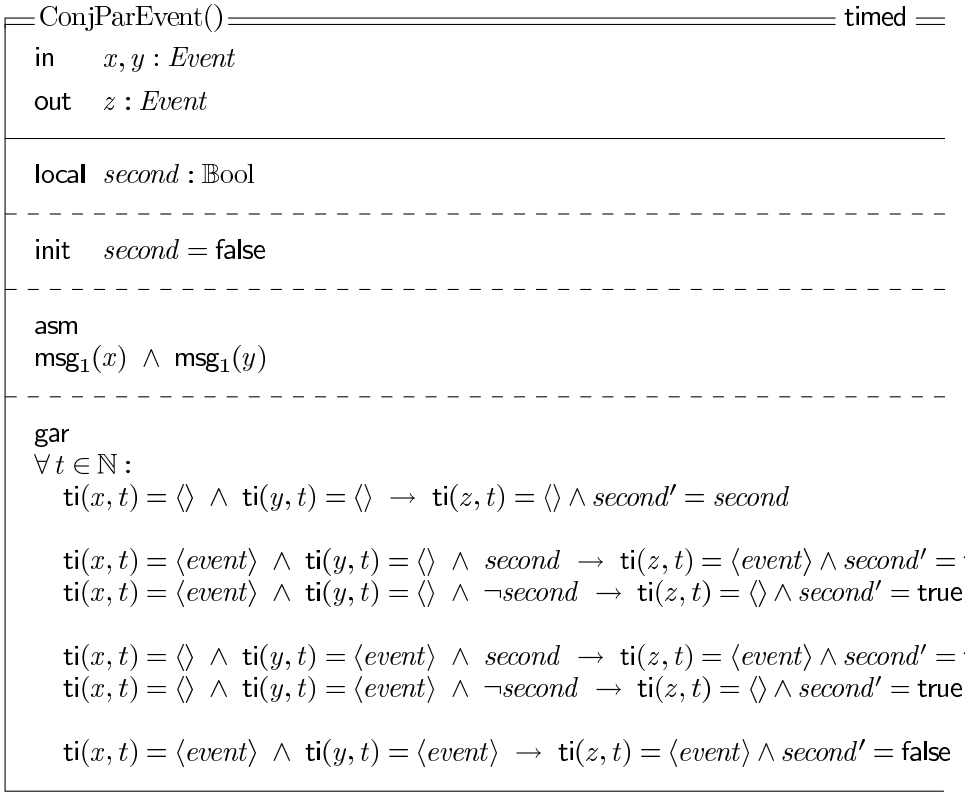
Figure 3: Simultaneously Composed Process $P \parallel Q$

Fig. 3 shows a graphical representation of parallel composition of two processes P and Q in general. The special components *ConjEvent* is specified below. We assume here, that the processes P and Q can be activated next time iff both of them are completed.

The sets of input, output and local channels are defined for processes and components as follows:

$$
\begin{aligned}
&Entry(P;\ Q) = entP\\
&Exit(P;\ Q) = extPQ\\
&I_{P;\,Q} = (I_P \setminus L_{P;\,Q}) \cup (I_Q \setminus L_{P;\,Q})\\
&O_{P;\,Q} = (O_P \setminus L_{P;\,Q}) \cup (O_Q \setminus L_{P;\,Q})\\
&L_{P;\,Q} = (I_P \cap O_Q) \cup (O_P \cap I_Q)
\end{aligned}
$$

$$
\begin{aligned}
I_{[P||Q]^{comp}} &= (I_{[P]^{comp}} \setminus L_{[P||Q]^{comp}}) \cup (I_{[Q]^{comp}} \setminus L_{[P||Q]^{comp}})\\
O_{[P||Q]^{comp}} &= (O_{[P]^{comp}} \setminus L_{[P||Q]^{comp}}) \cup (O_{[Q]^{comp}} \setminus L_{[P||Q]^{comp}})\\
L_{[P||Q]^{comp}} &= (I_{[P]^{comp}} \cap O_{[Q]^{comp}}) \cup (O_{[P]^{comp}} \cap I_{[Q]^{comp}})
\end{aligned}
$$



5.4 Specification Of An Repetitively Composed Process: Autonomous Version

In the autonomous version the entry point $Entry(P \circlearrowleft^A_{lpspec})$ as well as the exit point $Exit(P \circlearrowleft^A_{lpspec})$ are undefined for the black-box-view, because the process is started by themselves and repeated after the specified time.

$$
\begin{aligned}
&[P(i_1,\ldots,i_m,o_1,\ldots,o_n) \circlearrowleft^A_{lpspec}]^{comp} \;= \\
&\quad P^{spec}(entP, i_1,\ldots,i_m, extP, o_1,\ldots,o_n) \otimes \\
&\quad LoopSpec(extP, entP)
\end{aligned}
$$

Fig. 4 shows a graphical representation of this kind of loop composition. The special component *LoopSpec* can be defined in many ways according to the system needs. The important point is here that if the FOCUS process P (and correspondingly the FOCUS component P^{spec}) is only weak causal, then the component *LoopSpec* must be string causal, i.e. to have at least one time unit delay.

We present here a simple example of a loop-componet *LoopSpec*7 that restarts the process in 7 time units after it was complete.

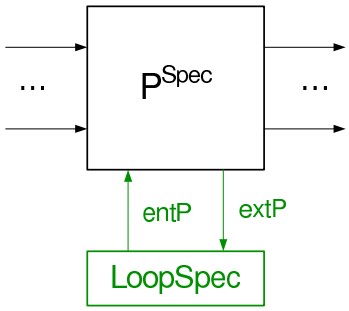


Figure 4: Repetitively Composed Process: Autonomous Version

LoopSpec7() timed

in $x : Event$

out $z : Event$

local $timer : \mathbb{N}$

init $timer = 0$

asm
$\mathsf{ts}(x)$

gar
$\mathsf{ti}(z, 0) = \langle event \rangle$

$\forall\, t \in \mathbb{N} :$

$timer = 0 \ \wedge \ \mathsf{ti}(x, t) = \langle\rangle \ \rightarrow \ \mathsf{ti}(z, t+1) = \langle\rangle \wedge timer' = 0$

$timer = 0 \ \wedge \ \mathsf{ti}(x, t) = \langle event \rangle \ \rightarrow \ \mathsf{ti}(z, t+1) = \langle\rangle \wedge timer' = 1$

$timer = 7 \ \rightarrow \ \mathsf{ti}(z, t+1) = \langle event \rangle \wedge timer' = 0$

$timer > 0 \ \wedge \ timer < 7 \ \rightarrow \ \mathsf{ti}(z, t+1) = \langle\rangle \wedge timer' = timer + 1$

The sets of input, output and local channels are defined for process and component as follows:

$Entry(P \circlearrowleft_{lpspec}) = \varnothing$
$Exit(P \circlearrowleft_{lpspec}) = \varnothing$
$I_{P \circlearrowleft^{A}_{lpspec}} = I_P$
$O_{P \circlearrowleft^{A}_{lpspec}} = O_P$
$L_{P \circlearrowleft^{A}_{lpspec}} = \varnothing$

$$I_{[P\circlearrowleft^A_{lpspec}]} = I_{[P]^{comp}} \setminus \{entP\}$$
$$O_{[P\circlearrowleft^A_{lpspec}]} = O_{[P]^{comp}} \setminus \{extP\}$$
$$L_{[P\circlearrowleft^A_{lpspec}]} = \{entP, extP\}$$

5.5 Specification Of An Repetitively Composed Process: Non-Autonomous Version

In the non-autonomous version the entry point $Entry(P\circlearrowleft^A_{lpspec})$ as well as the exit point $Exit(P\circlearrowleft^A_{lpspec})$ are defined for the black-box-view and are merged with the loop values:

$$[P(i_1,\ldots,i_m,o_1,\ldots,o_n)\circlearrowleft_{lpspec}]^{comp} =$$
$$P^{spec}(entP,i_1,\ldots,i_m,extP,o_1,\ldots,o_n)\otimes$$
$$LoopSpec(extP,entPL)\otimes$$
$$MergeEntryEvents(entPS,entPL,extP,entP)$$

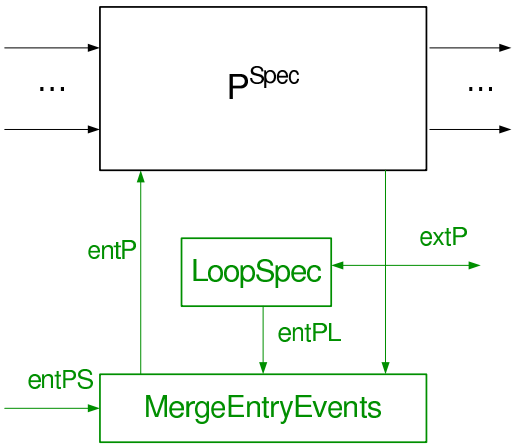


Figure 5: Repetitively Composed Process: Non-Autonomous Version

Fig. 5 shows a graphical representation of this kind of loop composition. The special component *LoopSpec* is defined in the similar way to the autonomous version.

The special component *MergeEntryEvents* can be defined in different ways to merge the start signals from outside and the start signals from the loop. The important point is here whether the process can be restarted before it was complete – we add to this component the input channel *extP* to check this. An example of a FOCUS specification for the component *MergeEntryEvents* is presented below.

The sets of input, output and local channels are defined for process and component as follows:

$Entry(P \circlearrowleft_{lpspec}) = entP$
$Exit(P \circlearrowleft_{lpspec}) = extP$
$I_{P\circlearrowleft_{lpspec}} = I_P$
$O_{P\circlearrowleft_{lpspec}} = O_P$
$L_{P\circlearrowleft_{lpspec}} = \varnothing$

$I_{[P\circlearrowleft_{lpspec}]} = (I_{[P]^{comp}} \cup \{entPS\}) \setminus \{entP\}$
$O_{[P\circlearrowleft_{lpspec}]} = O_{[P]^{comp}}$
$L_{[P\circlearrowleft_{lpspec}]} = \{entP, extP, entPL, entPS\}$

MergeEntryEvents() — timed

in $xS, xLoop, y : Event$

out $x : Event$

local $ready : \mathbb{B}\mathrm{ool}$

init $ready = \mathsf{false}$

asm
$\mathsf{ts}(xS) \ \wedge \ \mathsf{ts}(xLoop) \ \wedge \ \mathsf{ts}(y)$

gar
$\forall\, t \in \mathbb{N}:$
$\mathsf{ti}(xLoop, t) = \langle event \rangle \ \rightarrow \ \mathsf{ti}(x, t) = \langle event \rangle \wedge ready' = \mathsf{false}$

$\mathsf{ti}(xLoop, t) = \langle\rangle \wedge ready = \mathsf{false} \wedge \mathsf{ti}(xLoop, t) = \langle\rangle \ \rightarrow$
$\mathsf{ti}(x, t) = \langle\rangle \wedge ready' = \mathsf{false}$

$\mathsf{ti}(xLoop, t) = \langle\rangle \wedge ready = \mathsf{false} \wedge \mathsf{ti}(xLoop, t) = \langle event \rangle \ \rightarrow$
$\mathsf{ti}(x, t) = \langle\rangle \wedge ready' = \mathsf{true}$

$\mathsf{ti}(xLoop, t) = \langle\rangle \wedge ready = \mathsf{true} \wedge \mathsf{ti}(xS, t) = \langle\rangle \ \rightarrow$
$\mathsf{ti}(x, t) = \langle\rangle \wedge ready' = \mathsf{true}$

$\mathsf{ti}(xLoop, t) = \langle\rangle \wedge ready = \mathsf{true} \wedge \mathsf{ti}(xS, t) = \langle event \rangle \ \rightarrow$
$\mathsf{ti}(x, t) = \langle event \rangle \wedge ready' = \mathsf{true}$

5.6 Special Cases

In Sections 5.1–5.3 it doesn't matter whether the set $I_P \cap I_Q$ is empty or not, because we can simply split the input streams to many components. But for all composed specifications presented above we need to assume that the set $O_P \cap O_Q$ is empty, because we cannot simply join the output streams, we need to merge them using a special component, e.g. the component *Merge* specified in Section 5.2.

Thus, for the case $(O_P \cap O_Q) \neq \varnothing$ we need to redefine the composition equality.

Assume that $(O_P \cap O_Q) = \{o_1, \ldots, o_l\}$ for some l, s.t. $1 \leq l \leq n$ and $l \leq z$, i.e. $O_P = o_1, \ldots, o_l, o_{l+1}, \ldots, o_n$ and $O_Q = o_1, \ldots, o_l, y_{l+1}, \ldots, y_n$ (the data types of $o_1, \ldots, o_l$ are $M_1, \ldots, M_n$ respectively). Then

$$
\begin{aligned}
&[P(i_1, \ldots, i_m, o_1, \ldots, o_n);\ Q(x_1, \ldots, x_k, y_1, \ldots, y_z)]^{comp} \ = \\
&\quad P^{spec}(entP, i_1, \ldots, i_m, extP, o_1, \ldots, o_n) \otimes \\
&\quad Q^{spec}(extP, x_1, \ldots, x_k, extQ, y_1, \ldots, y_z) \otimes \\
&\quad Merge(M_1)(o_1, y_1, oy_1) \otimes \cdots \otimes Merge(M_l)(o_l, y_l, oy_l)
\end{aligned}
$$

$$
\begin{aligned}
&[P(i_1, \ldots, i_m, o_1, \ldots, o_n) \oplus Q(x_1, \ldots, x_k, y_1, \ldots, y_z)]^{comp} \ = \\
&\quad P^{spec}(entP, i_1, \ldots, i_m, extP, o_1, \ldots, o_n) \otimes \\
&\quad Q^{spec}(extP, x_1, \ldots, x_k, extQ, y_1, \ldots, y_z) \otimes \\
&\quad GuardEvent(entPQ, entP, entQ) \otimes \\
&\quad Merge(Event)(extP, extQ, extPQ) \otimes \\
&\quad Merge(M_1)(o_1, y_1, oy_1) \otimes \cdots \otimes Merge(M_l)(o_l, y_l, oy_l)
\end{aligned}
$$

$$
\begin{aligned}
&[P(i_1, \ldots, i_m, o_1, \ldots, o_n) \parallel Q(x_1, \ldots, x_k, y_1, \ldots, y_z)]^{comp} \ = \\
&\quad P^{spec}(entP, i_1, \ldots, i_m, extP, o_1, \ldots, o_n) \otimes \\
&\quad Q^{spec}(entP, x_1, \ldots, x_k, extQ, y_1, \ldots, y_z) \otimes \\
&\quad ConjParEvent(extP, extQ, extPQ) \otimes \\
&\quad Merge(M_1)(o_1, y_1, oy_1) \otimes \cdots \otimes Merge(M_l)(o_l, y_l, oy_l)
\end{aligned}
$$

6 Case Study: Pumping Station

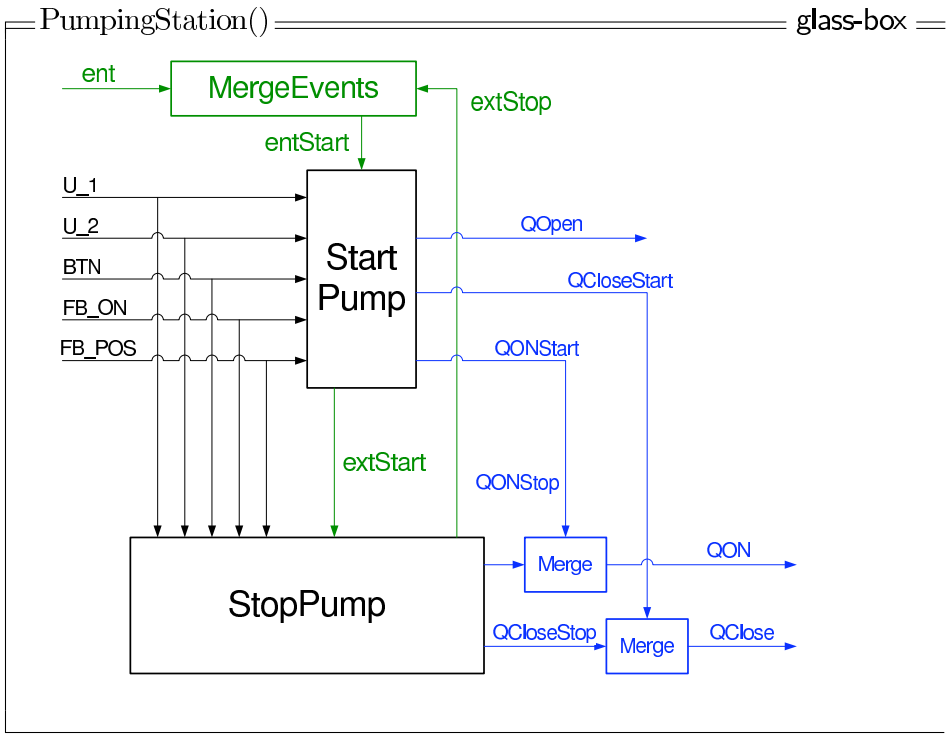
In this section a case study to the approach is presented: we specify in FOCUS the main processes of a pumping station. Using such kind of specification we can verify properties of pumping station in a formal way, e.g. by translating the FOCUS specifications to Isabelle/HOL and using the Isabelle tool to make the proofs.

6.1 Data Types and Constants

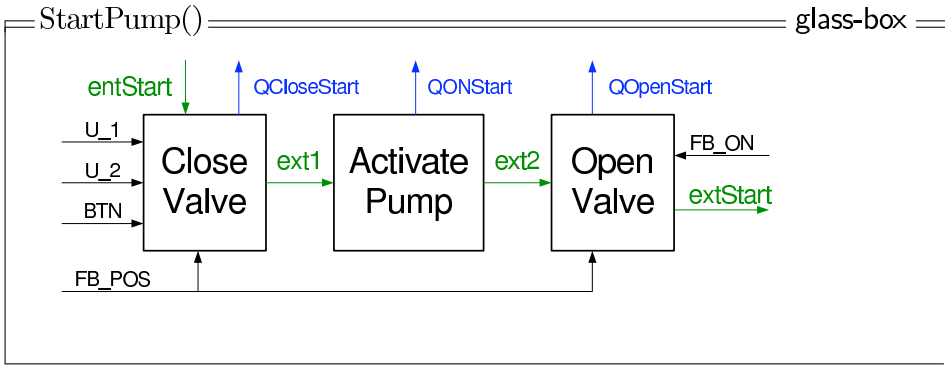
The following user-defined data types and constants will be used within the case study:

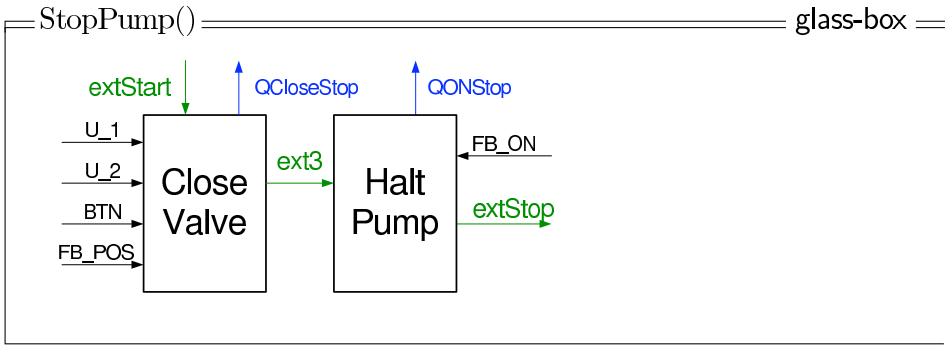
type $CloseValveStates = \{Open, Closing, Closed\}$
type $HaltPumpStates = \{On, Halting, Off\}$
type $ActivatePumpStates = \{On, Off\}$
type $OpenValveStates = \{Wait, Opening, Open, On\}$
const $ValveClosed \in \mathbb{N}$; $ValveClosed = 100$
const $ValveOpen \in \mathbb{N}$; $ValveClosed = 0$
const $OpenValveDelay \in \mathbb{N}$; $ValveClosed = 10$

6.2 System Architecture



The system process *PumpingStation* consists of two subprocesses, *StartPump* and *StopPump*, building an activation loop as shown above. Each of these two subprocesses is non-elementary, i.e. sequentially composed of a number of elementary processes.





Specifications of elementary processes *CloseValse*, *ActivatePump*, *OpenValve*, and *HaltPump* as well as the corresponding FOCUS specification are discussed in the following subsections. All these processes are strong causal with delay of one time unit.

Please note that we don't need here any buffers (according to the behavior of the process) for input channels for any of the elementary processes. Please also note that all these processes are strongly-causal.

6.3 CloseValve Component

process CloseValve() — **timed**

entry *start*

exit *ext*

---

in $U_1, U_2, Pos : \mathbb{N};\ Btn : \mathbb{B}\text{ool}$

out $Close : \mathbb{B}\text{ool}$

---

local $active : \mathbb{B}\text{ool};\ CloseValveSt \in CloseValveStates$

---

init $active = \mathsf{false};\ CloseValveSt = Open$

---

initProcess $CloseValveSt = Open$

---

asm

$\mathsf{msg}_1(U_1) \ \wedge\ \mathsf{msg}_1(U_2) \ \wedge\ \mathsf{msg}_1(Pos) \ \wedge\ \mathsf{msg}_1(Btn)$

---

gar

0 $\quad \mathsf{ti}(ext, 0) = \langle\rangle \ \wedge\ \mathsf{ti}(Close, 0) = \langle\rangle$

1 $\quad CloseValveSt = Closing \ \wedge\ \mathsf{ti}(Pos, t) = \langle ValveClosed\rangle \ \rightarrow$
$CloseValveSt' = Closed \ \wedge\ \mathsf{ti}(Close, t+1) = \langle\rangle \ \wedge$
$\mathsf{ti}(ext, t+1) = \langle event\rangle \ \wedge\ active' = \mathsf{false}$

2 $\quad CloseValveSt = Open \ \wedge$
$\mathsf{ti}(U_1, t) = \langle x\rangle \ \wedge\ x \geq TankLevelMin \ \wedge\ \mathsf{ti}(U_2, t) = \langle y\rangle \ \wedge\ y \leq TankLevelMax \ \wedge$
$\mathsf{ti}(Btn, t) = \langle\mathsf{true}\rangle \ \rightarrow$
$CloseValveSt' = Closing \ \wedge\ \mathsf{ti}(Close, t+1) = \langle\mathsf{true}\rangle \ \wedge$
$\mathsf{ti}(ext, t+1) = \langle\rangle \ \wedge\ active' = \mathsf{true}$

3 $\quad CloseValveSt = Open \ \wedge$
$\neg(\mathsf{ti}(U_1, t) = \langle x\rangle \ \wedge\ x \geq TankLevelMin \ \wedge\ \mathsf{ti}(U_2, t) = \langle y\rangle \ \wedge\ y \leq TankLevelMax \ \wedge$
$\mathsf{ti}(Btn, t) = \langle\mathsf{true}\rangle) \ \rightarrow$
$CloseValveSt' = Open \ \wedge\ \mathsf{ti}(Close, t+1) = \langle\rangle \ \wedge$
$\mathsf{ti}(ext, t+1) = \langle\rangle \ \wedge\ active' = \mathsf{true}$

4 $\quad CloseValveSt = Closing \ \wedge\ \mathsf{ti}(Pos, t) \neq \langle ValveClosed\rangle \ \rightarrow$
$CloseValveSt' = Closing \ \wedge\ \mathsf{ti}(Close, t+1) = \langle\rangle \ \wedge$
$\mathsf{ti}(ext, t+1) = \langle\rangle \ \wedge\ active' = \mathsf{true}$

We can prove that this specification implies that if the process is active, it can't be in state *Closed*:

$$active = \mathsf{true} \rightarrow CloseValveSt \neq Closed \qquad (*)$$

According to the FOCUS extension presented in Section 2 the optimized version

CloseValveOptim of the specification will have in the guarantee part two formulas less then in the non-optimized version (the 3rd and the 4th formula will be covered by the semantics given by the specification label *optimized*):

process CloseValveOptim() — timed, optimized

entry *start*

exit *ext*

in $U_1, U_2, Pos : \mathbb{N};\ Btn : \mathbb{B}ool$

out $Close : \mathbb{B}ool$

local $active : \mathbb{B}ool;\ CloseValveSt \in CloseValveStates$

init $active = \mathsf{false};\ CloseValveSt = Open$

initProcess $CloseValveSt = Open$

asm

$\mathsf{msg}_1(U_1) \wedge \mathsf{msg}_1(U_2) \wedge \mathsf{msg}_1(Pos) \wedge \mathsf{msg}_1(Btn)$

gar

0 $\mathsf{ti}(ext, 0) = \langle\rangle \wedge \mathsf{ti}(Close, 0) = \langle\rangle$

1 $CloseValveSt = Closing \wedge \mathsf{ti}(Pos, t) = \langle ValveClosed\rangle \rightarrow$
$CloseValveSt' = Closed \wedge \mathsf{ti}(Close, t+1) = \langle\rangle \wedge$
$\mathsf{ti}(ext, t+1) = \langle event\rangle \wedge active' = \mathsf{false}$

2 $CloseValveSt = Open \wedge$
$\mathsf{ti}(U_1, t) = \langle x\rangle \wedge x \geq TankLevelMin \wedge \mathsf{ti}(U_2, t) = \langle y\rangle \wedge y \leq TankLevelMax \wedge$
$\mathsf{ti}(Btn, t) = \langle\mathsf{true}\rangle \rightarrow$
$CloseValveSt' = Closing \wedge \mathsf{ti}(Close, t+1) = \langle\mathsf{true}\rangle \wedge$
$\mathsf{ti}(ext, t+1) = \langle\rangle \wedge active' = \mathsf{true}$

A specification *CloseValvePredicates* of the same process *CloseValve* done using predicates *CloseValveEndingCondition* and *Calculations* is presented below.

CloseValveEndingCondition

$CvState \in CloseValveStates;\ p \in \mathbb{N}^*$

$CvState = Closing \wedge p = \langle ValveClosed\rangle$

CloseValveCalculations

$a, b, btn, pos \in \mathbb{N}^*$; $CvState, CvStateNext \in CloseValveStates$; $cl \in \mathbb{B}ool^*$

$CvState = Open \ \wedge$
$a = \langle x \rangle \ \wedge \ x \geq TankLevelMin \ \wedge \ b = \langle y \rangle \ \wedge \ y \leq TankLevelMax \ \wedge \ btn = \langle \mathsf{true} \rangle$
$\quad \rightarrow \ CvStateNext = Closing \ \wedge \ cl = \langle \mathsf{true} \rangle$
$\wedge$
$CvState = Open \ \wedge$
$\neg(a = \langle x \rangle \ \wedge \ x \geq TankLevelMin \ \wedge \ b = \langle y \rangle \ \wedge \ y \leq TankLevelMax \ \wedge \ btn = \langle \mathsf{true} \rangle)$
$\quad \rightarrow \ CvStateNext = Open \ \wedge \ cl = \langle\rangle$
$\wedge$
$CvState = Closing \ \wedge \ pos \neq \langle ValveClosed \rangle \ \rightarrow \ CvStateNext = Closing \ \wedge \ cl = \langle\rangle$

---

process CloseValvePredicates() timed

entry *start*

exit *ext*

- -

in $U_1, U_2, Pos : \mathbb{N}$; $Btn : \mathbb{B}ool$

out $Close : \mathbb{B}ool$

---

local $active : \mathbb{B}ool$; $CloseValveSt \in CloseValveStates$

- -

init $active = \mathsf{false}$; $CloseValveSt = Open$

- -

initProcess $CloseValveSt = Open$

- -

asm
$\mathsf{msg}_1(U_1) \ \wedge \ \mathsf{msg}_1(U_2) \ \wedge \ \mathsf{msg}_1(Pos) \ \wedge \ \mathsf{msg}_1(Btn)$

- -

gar

0 $\mathsf{ti}(ext, 0) = \langle\rangle \ \wedge \ \mathsf{ti}(Close, 0) = \langle\rangle$

1 $CloseValveEndingCondition(CloseValveSt, \mathsf{ti}(Pos, t)) \ \rightarrow$
$CloseValveCalculationsF(CloseValveSt', \mathsf{ti}(Close, t+1)) \ \wedge$
$\mathsf{ti}(ext, t+1) = \langle event \rangle \ \wedge \ active' = \mathsf{false}$

2 $\neg CloseValveEndingCondition(CloseValveSt, \mathsf{ti}(Pos, t)) \ \rightarrow$
$CloseValveCalculations(\mathsf{ti}(U_1, t), \mathsf{ti}(U_2, t), \mathsf{ti}(Btn, t), \mathsf{ti}(Pos, t),$
$CloseValveSt, CloseValveSt', \mathsf{ti}(Close, t+1)) \ \wedge$
$\mathsf{ti}(ext, t+1) = \langle\rangle \ \wedge \ active' = \mathsf{true}$

The constant $ValveClosed \in \mathbb{N}$ is defined in Section 6.1.

From this definition follows also

$$\neg CloseValveEndingCondition(CloseValveSt, \mathsf{ti}(Pos, t)) =$$
$$(CloseValveSt \neq Closing \ \lor \ \mathsf{ti}(Pos, t) \neq \langle ValveClosed \rangle)$$

CloseValveCalculationsF

$$CvStateNext \in CloseValveStates; \ cl \in \mathbb{B}ool^*$$

$$CvStateNext = Closed \ \land \ cl = \langle\rangle$$

The specifications *CloseValvePredicates* an *CloseValve* are (sematically) equivalent: it is easy to see that the interface and assumption parts of these specifications are exactly the same, the only difference is in the guarantee part, but this difference is only syntactical.

The first formulas of both specification are equivalent according to the definitions of the predicates *CloseValveEndingCondition* and *CloseValveCalculationsF*.

The second formula of *CloseValvePredicates* is equivalent to the conjunction of the second, the third and the fourth formulas of *CloseValve*:

$$\begin{array}{l}
\neg CloseValveEndingCondition(CloseValveSt, \mathsf{ti}(Pos, t)) \ \rightarrow \\
\quad CloseValveCalculations(\mathsf{ti}(U_1, t), \mathsf{ti}(U_2, t), \mathsf{ti}(Btn, t), \mathsf{ti}(Pos, t), \\
\qquad\qquad\qquad\qquad CloseValveSt, CloseValveSt', \mathsf{ti}(Close, t+1)) \ \land \\
\quad \mathsf{ti}(ext, t) = \langle\rangle \ \land \ active' = \mathsf{true} \\
\equiv \\
\neg(CloseValveSt = Closing \ \land \ \mathsf{ti}(Pos, t) = \langle ValveClosed \rangle) \ \rightarrow \\
(CloseValveSt = Open \ \land \ \mathsf{ti}(U_1, t) = \langle x \rangle \ \land \ x \geq TankLevelMin \ \land \\
\quad \mathsf{ti}(U_2, t) = \langle y \rangle \ \land \ y \leq TankLevelMax \ \land \ \mathsf{ti}(Btn, t) = \langle \mathsf{true} \rangle \ \rightarrow \\
\quad CloseValveSt' = Closing \ \land \ \mathsf{ti}(Close, t+1) = \langle \mathsf{true} \rangle \\
\land \\
CloseValveSt = Open \ \land \ \neg(\mathsf{ti}(U_1, t) = \langle x \rangle \ \land \ x \geq TankLevelMin \ \land \\
\quad \mathsf{ti}(U_2, t) = \langle y \rangle \ \land \ y \leq TankLevelMax \ \land \ \mathsf{ti}(Btn, t) = \langle \mathsf{true} \rangle) \ \rightarrow \\
\qquad CloseValveSt' = Open \ \land \ \mathsf{ti}(Close, t+1) = \langle\rangle \\
\land \\
CloseValveSt = Closing \ \land \ \mathsf{ti}(Pos, t) \neq \langle ValveClosed \rangle \ \rightarrow \\
\qquad CloseValveSt' = Closing \ \land \ \mathsf{ti}(Close, t+1) = \langle\rangle) \ \land \\
\quad \mathsf{ti}(ext, t) = \langle\rangle \ \land \ active' = \mathsf{true}
\end{array}$$

$$
\begin{array}{l}
\equiv \\
\mathit{CloseValveSt} = \mathit{Open} \ \wedge \ \mathsf{ti}(U_1, t) = \langle x \rangle \ \wedge \ \mathsf{ti}(U_2, t) = \langle Y \rangle \ \wedge \ \mathsf{ti}(\mathit{Btn}, t) = \langle \mathsf{true} \rangle \ \rightarrow \\
\quad \mathit{CloseValveSt}' = \mathit{Closing} \ \wedge \ \mathsf{ti}(\mathit{Close}, t+1) = \langle \mathsf{true} \rangle \ \wedge \quad \mathsf{ti}(\mathit{ext}, t) = \langle\rangle \ \wedge \ \mathit{active}' = \mathsf{true} \\
\wedge \\
\mathit{CloseValveSt} = \mathit{Open} \ \wedge \ \neg(\mathsf{ti}(U_1, t) = \langle x \rangle \ \wedge \ \mathsf{ti}(U_2, t) = \langle Y \rangle \ \wedge \ \mathsf{ti}(\mathit{Btn}, t) = \langle \mathsf{true} \rangle) \ \rightarrow \\
\quad \mathit{CloseValveSt}' = \mathit{Open} \ \wedge \ \mathsf{ti}(\mathit{Close}, t+1) = \langle\rangle \ \wedge \quad \mathsf{ti}(\mathit{ext}, t) = \langle\rangle \ \wedge \ \mathit{active}' = \mathsf{true} \\
\wedge \\
\mathit{CloseValveSt} = \mathit{Closing} \ \wedge \ \mathsf{ti}(\mathit{Pos}, t) \neq \langle \mathit{ValveClosed} \rangle \ \rightarrow \\
\quad \mathit{CloseValveSt}' = \mathit{Closing} \ \wedge \ \mathsf{ti}(\mathit{Close}, t+1) = \langle\rangle \ \wedge \quad \mathsf{ti}(\mathit{ext}, t) = \langle\rangle \ \wedge \ \mathit{active}' = \mathsf{true}
\end{array}
$$

Please note that by definition the guarantee part the process specification describes the situation for the case $\mathit{active} = \mathsf{true}$. According to $(*)$ this means that $\mathit{CloseValveSt} \neq \mathit{Closed}$, i.e. $\mathit{CloseValveSt} = \mathit{Open} \vee \mathit{CloseValveSt} = \mathit{Closing}$.

6.4 ActivatePump Component

process ActivatePump() — **timed**

entry *start*

exit *stop*

---

out $\mathit{QOn} : \mathbb{B}\mathrm{ool}$

---

local $\mathit{active} : \mathbb{B}\mathrm{ool}$; $\mathit{ActivatePumpSt} \in \mathit{ActivatePumpStates}$

---

init $\mathit{active} = \mathsf{false}$; $\mathit{ActivatePumpSt} = \mathit{Off}$

---

initProcess $\mathit{ActivatePumpSt} = \mathit{Off}$

---

asm
true

---

gar
1 $\mathit{ActivatePumpSt} = \mathit{Off} \ \rightarrow$
$\mathit{ActivatePumpSt}' = \mathit{On} \ \wedge \ \mathsf{ti}(\mathit{QOn}, t+1) = \langle \mathsf{false} \rangle \ \wedge$
$\mathsf{ti}(\mathit{stop}, t+1) = \langle \mathit{event} \rangle \ \wedge \ \mathit{active}' = \mathsf{false}$

6.5 OpenValve Component

process OpenValve() — timed

entry *start*

exit *ext*

---

in $FB_{On} : \mathbb{B}\text{ool};\ FB_{Pos} : \mathbb{N}$

out $QOpen : \mathbb{B}\text{ool}$

---

local $active : \mathbb{B}\text{ool};\ OpenValveSt \in OpenValveStates;\ timer : \mathbb{N}$

---

init $active = \text{false};\ OpenValveSt = Wait;\ timer = 0$

---

initProcess $OpenValveSt = Wait;\ timer = 0$

---

asm

$\text{msg}_1(FB_{On}) \ \wedge\ \text{msg}_1(FB_{On})$

---

gar

0 $\text{ti}(ext, 0) = \langle\rangle \ \wedge\ \text{ti}(QOpen, 0) = \langle\rangle$

1 $OpenValveSt = Wait \ \wedge\ timer < OpenValveDelay \ \rightarrow$
$OpenValveSt' = Wait \ \wedge\ timer' = timer + 1 \ \wedge\ \text{ti}(QOpen, t+1) = \langle\rangle \ \wedge$
$\text{ti}(ext, t+1) = \langle\rangle \ \wedge\ active' = \text{true}$

2 $OpenValveSt = Wait \ \wedge\ timer = OpenValveDelay \ \rightarrow$
$OpenValveSt' = Opening \ \wedge\ timer' = 0 \ \wedge\ \text{ti}(QOpen, t+1) = \langle\text{true}\rangle \ \wedge$
$\text{ti}(ext, t+1) = \langle\rangle \ \wedge\ active' = \text{true}$

3 $OpenValveSt = Opening \ \wedge$
$\text{ti}(FB_{Pos}, t) = \langle ValveOpen\rangle \ \rightarrow$
$OpenValveSt' = Open \ \wedge\ \text{ti}(QOpen, t+1) = \langle\rangle \ \wedge$
$\text{ti}(ext, t+1) = \langle\rangle \ \wedge\ active' = \text{true}$

4 $OpenValveSt = Open \ \wedge$
$\text{ti}(FB_{On}, t) = \langle\text{true}\rangle \ \rightarrow$
$OpenValveSt' = On \ \wedge\ \text{ti}(QOpen, t+1) = \langle\rangle \ \wedge$
$\text{ti}(ext, t+1) = \langle event\rangle \ \wedge\ active' = \text{false}$

5 $OpenValveSt = Opening \ \wedge$
$\text{ti}(FB_{Pos}, t) \neq \langle ValveOpen\rangle \ \rightarrow$
$OpenValveSt' = Opening \ \wedge\ \text{ti}(QOpen, t+1) = \langle\rangle \ \wedge$
$\text{ti}(ext, t+1) = \langle\rangle \ \wedge\ active' = \text{true}$

According to the FOCUS extension presented in Section 2 the optimized version *OpenValveOptim* of the specification will have in the guarantee part one formula less then in the non-optimized version (the 5th formula will be covered by the semantics given by the specification label *optimized*).

We can prove that this specification implies that if the process is active, it can't be in state *Off*:

$$active = \mathsf{true} \rightarrow OpenValveSt \neq On$$

We can specify the process *OpenValve* also using predicates *OpenValveCalculations* and *OpenValveEndingCondition*.

6.6 HaltPump Component

process HaltPump — timed

entry *start*

exit *stop*

---

in $FB_{On} : \mathbb{B}\mathrm{ool}$

out $QOn : \mathbb{B}\mathrm{ool}$

---

local $active : \mathbb{B}\mathrm{ool}$; $HaltPumpSt \in HaltPumpStates$

---

init $active = \mathsf{false}$; $HaltPumpSt = On$

---

initProcess $HaltPumpSt = On$

---

asm

$\mathsf{msg}_1(FB_{On})$

---

gar

1 $HaltPumpSt = On \rightarrow$
$HaltPumpSt' = Halting \ \wedge \ \mathsf{ti}(QOn, t+1) = \langle \mathsf{true} \rangle$

2 $HaltPumpSt = Halting \ \wedge$
$\mathsf{ti}(FB_{On}, t) = \langle \mathsf{false} \rangle \rightarrow$
$HaltPumpSt' = Off \ \wedge \ \mathsf{ti}(QOn, t+1) = \langle\rangle$

3 $HaltPumpSt = Halting \ \wedge$
$\mathsf{ti}(FB_{On}, t) \neq \langle \mathsf{false} \rangle \rightarrow$
$HaltPumpSt' = Halting \ \wedge \ \mathsf{ti}(QOn, t+1) = \langle\rangle$

According to the FOCUS extension presented in Section 2 the optimized version *HaltPumpOptim* of the specification will have in the guarantee part one formula less then in the non-optimized version (the 3rd formula will be covered by the semantics given by the specification label *optimized*).

We can prove that this specification implies that if the process is active, it can't be in state *Off*:

$$active = \mathsf{true} \rightarrow HaltPumpSt \neq Off$$

We can specify the process *HaltPump* also using predicates *HaltPumpCalculations* and *HaltPumpEndingCondition*.

7 Conclusions

Specifying components and system in a formal language is helpful ho have a possibility to present within the language also such a concept as *process*. This paper presents the corresponding extension of the formal specification language FOCUS [1] as well as of the methodology *Focus on Isabelle* [4] by the process language: how an elementary and a composed process can be specified with the FOCUS language, which properties have different kinds of composition operators and how a FOCUS process can be represented by a FOCUS component.

As the starting point of the process language a formal model for specification and analysis of work flows [2] was taken.

Another topic covered in this paper is optimization of the FOCUS language to specify some trivial cases implicitly, by the specification semantics.

Acknowledgments

I would like to thank C. Leuxner for numerous discussions on the subject of this paper.

References

[1] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[2] C. Leuxner, W. Sitou, and B. Spanfelner. A formal model for work flows. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 135–144. IEEE, 2010.

[3] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[4] M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, TU München, 2007.

[5] M. Spichkova. User Guide for the FOCUS representation in LaTeX. Technical Report TUM, TU München, 2011.