3. Reliability and Availability Aspects: Fault-Tolerance

3.1 Overview

In this chapter we present the design and the verification of a fault-tolerant client-server configuration. Dependability of the server is achieved by means of a *dual computer system* operating in a modified *hot standby mode*. The specification and verification covers normal behaviour, recovery after failure and re-integration of a repaired component into the running system. The results described are an extension of the work published in [73, 74, 75, 76].

In Section 3.2 a development project at Philips is sketched which originally motivated the investigations presented here. The rest of this chapter is divided into two parts: Section 3.3 describes the development and the verification of the fault-tolerant system and Section 3.4 presents a formal justification of a proof method applied in the case study.

Fault-tolerant systems have been investigated by many authors (J. Rushby and F. Christian present overviews in [102] and [15]) using different specification styles and verification approaches. To name some characteristic examples, Christian [13, 14] was one of the first to introduce rigorous approaches for reasoning about fault-tolerance. Schlichting and Schneider [105] first analysed the concept of fail-stop processors which is an important building block for the design of fault-tolerant systems and also used as a prerequisite to the fault-tolerant dual computer system introduced in this chapter. Jalote [49] uses informal presentation style, Dijkstra [23] introduced the technique of self-stabilising algorithms, He Jifeng and Tony Hoare [51] presented a fault-tolerant master-standby solution verified purely by means of algebraic reasoning in the CSP process algebra, Nordahl [70] and Schepers [104] applied behavioural specifications and compositional reasoning and Zhiming, Sørensen, Ravn and Chaochen [118] used duration calculus in combination with probabilistic methods.

The objective of our contribution is twofold:

- Show the suitability of the framework of Chapter 2 by structuring the design and verification steps according to the standardised development procedure.
- While the authors cited above mainly used a single style of reasoning, we wish to demonstrate that the complex verification obligations that have to be mastered are best solved by the application of *several* specification and verification formalisms.

To meet these objectives we proceed as follows: In Section 3.3.1 the requirements for the faul-tolerant server system are described in an informal way. Section 3.3.2 presents the details of the implementation. This will be helpful to understand the systematic top-down development with accompanying correctness proofs in the sections to follow. In Section 3.3.3 the formal system requirements are specified. Section 3.3.4 describes the architecture on system level and introduces the normal and acceptable behaviour specifications of the sub-systems created during this decomposition step. The verification on system level is performed in Section 3.3.5. As an alternative to the techniques applied in Chapter 2, safety properties and deadlock freedom are now shown in one step, using the compositional proof theory of CSP in the failures-divergence model. In Section 3.3.6 the sub-system design is described. For the dual computer, which is a sub-system of the complete server, another new technique for top-down design and associated verification is introduced: We define two explicit sequential CSP processes, modelling the normal behaviour and the acceptable behaviour of the dual computer, as far as visible at interface level. For sequential processes, behavioural properties can be verified by transforming the process into a sequential nondeterministic program and proving certain invariant properties about the latter. This concept is based on a theory of Apt and Olderog [5] and has already been informally applied in [74]. In Section 3.4 the complete formal justification will be presented, linking the denotational semantics of the failures model of CSP with the operational semantics of sequential nondeterministic programs. The technique is applied to prove that the two sequential processes satisfy the normal and acceptable behaviour specifications introduced for the dual computer on system level. In a second step the explicit internal process components of the dual computer are verified by means of model checking against the two sequential processes. This proves that the detailed design of the dual computer really meets its behavioural specification, making it superfluous to associate behavioural specifications with the internal processes and to verify them with the compositional techniques used on system level.

3.2 Related Industrial Projects

In 1984 Philips started the development of a distributed information and dispatching system for German police forces. Due to the dependability requirements of the customer, the critical user information had to be managed on a fault-tolerant database server, guaranteeing nearly non-stop availability for this data and its associated access and manipulation functions. Though at that time many of the important concepts about distributed systems were already available (see, for example, [55]), 1984 still has to be regarded as the "Stone Age" of distributed systems: The computer manufacturer offered ETHERNET hardware to build a local area network, but provided communication software only for the OSI layers 1 and 2. Therefore my team at Philips was responsible for the development of a distributed operating system on top of these layers, implementing the "essentials" of the OSI layers 3 to 6, which were necessary for the application. In addition, our task was to design and implement the fault-tolerance mechanisms for the database server, to be realised as a fault-tolerant dual computer system.

The customer's overall dependability requirements and the complexity of the application software led to the following basic requirements for the dual computer system:

- It should be unnecessary to incorporate specific software constructs in the application software in order to react properly on the failure of one computer. The whole set of control and synchronisation mechanisms needed to implement the dual computer system should be encapsulated in a subordinate service layer.
- A crash of one computer should be tolerated without any data losses or specific recovery

measures to be triggered on other computers in the network.

- The re-integration of a repaired computer should be possible without having to stop the system.

A transient failure corrupting isolated data bytes in one computer was considered as non-critical, because the data consisted mostly of natural-language texts. Therefore it was not necessary to use a higher degree of redundancy and compare inputs and outputs by means of voters.

These considerations led to the following design concept (for more details see [73]): Both computers were implemented as independent nodes in the network, equipped with their own local database. In normal operation, every job transmitted on the network to the database server was received by both computers, making use of the multi addressing capabilities of the ETHERNET. Transactions changing the database were executed on both computers. To this end, the computers synchronised by means of an algorithm which guaranteed equivalent serialisations of transactions on both local data bases. Read-only transactions were only executed on one computer while the other stored the associated job until the result had been successfully returned to the client. If several read-only jobs had to be processed in parallel, they were distributed between both computers. This transaction-oriented synchronisation concept reduced the amount of control messages to be exchanged, and the distribution of read-only transactions made up for the overhead caused by the synchronisation mechanism.

If one of the computers failed, the other one could continue without delay, since it always possessed the actual state of data in its local database. Therefore recovery measures only consisted in the re-start of read-only transactions which had been started, but not completed on the crashed computer. Both computers used dual-ported mirrored disks to store their local database information. For re-integration, the active computer shared one of its disks with the repaired component. Using an atomic read-write operation provided by the disk driver, all disks could be brought into an consistent state without stopping the application.

The mechanisms for synchronisation and re-integration of computers were quite complex, and first experiences with a prototype implementation implied that conventional testing and inspection methods would be insufficient to ensure the correct operation of the dual computer system. Therefore formal code verification was performed, using Hoare Logic for Pascal programs. The work described below was a result of the efforts to verify the correctness of the mechanisms on a more abstract level, ensuring that the algorithms applied could really guarantee the fault-tolerance requirements. The practical application of formal specification and verification methods indicated that it would be helpful if instead of a single formal methods *several* methods could be applied on different sub-tasks and on different stages of the development process. This motivated the combined use of CSP and verification techniques for nondeterministic sequential programs described in [74] and formalised in Section 3.4.

3.3 Case Study: Fault-Tolerant Server System With Repair

3.3.1 Informal Problem Description

Normal Behaviour The objective of the case study is to develop a fault-tolerant server system SYS for the client-server configuration depicted in Figure 3.1. The task of SYS is to read inputs x from the $CLIENT$ application on channel app_tx and return the result of a computation $y := f(x)$ on the receive channel app_rc. It is required that the *normal behaviour* should satisfy

1. SYS returns results in the order of the corresponding inputs.
2. To guarantee predictable response time, every acceptable implementation of SYS delivers a result after a globally bounded number of internal events following the corresponding input.
3. Analogously, SYS always accepts another input after a globally bounded number of internal events following the preceding input.

In these requirements, *"globally bounded"* means that the response time determined by the number of internal events between input and output is bounded by a constant which does not depend on the specific executions performed[1].

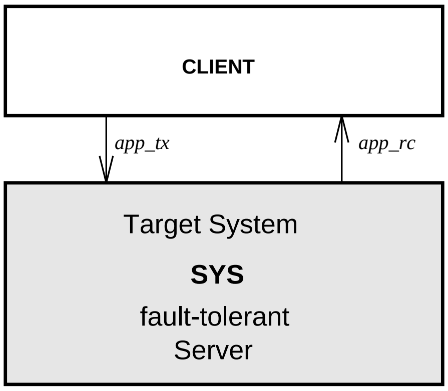

Figure 3.1: Client-server system.

Threat Analysis and Design Considerations It is assumed that external threats may be disregarded and that SYS is only threatened by internal hardware failure, provided that the software has been implemented correctly. Furthermore we assume the availability of *fail-stop* computers that either deliver a computation result correctly or stop completely[2]. We

[1]Note that this implies non-divergence of SYS and is a stronger requirement than the predicate $NODIV$ introduced in [44, p. 125], stating that the internal events are bounded by a function of the number of visible events already produced.

[2]Fail-stop components can be designed by means of duplicated synchronised CPUs, each using its own local memory. The write operations of the CPUs are compared, and the system is stopped as soon as these operations differ. Using this technique has the disadvantage that duplicated hardware is needed to construct one fail-stop component, so at least 4-modular redundancy is needed to construct a fault-tolerant system. Moreover, this only provides fail-stop functionality on hardware level and requires that the same software

assume the fault hypothesis that the probability for the second computer to fail while the first is still being repaired after a preceding crash may be neglected. Therefore a hardware platform consisting of two fail-stop components will suffice to meet the reliability requirements of the user.

Acceptable Behaviour For the *acceptable behaviour* $SYS \wr \Delta_{SYS}$ (at least one computer operates normally) it is required that $SYS \wr \Delta_{SYS}$ shows the same behaviour as SYS at the client interface. In addition to the requirements listed above this means that

1. a failure of one computer does not become visible at the interface to $CLIENT$,
2. a repaired computer may be re-integrated without visible effects at the interface to $CLIENT$.

Specifically, it is required for acceptable behaviour that a globally bounded response time can also be guaranteed in the presence of failures and re-integration activities.

3.3.2 Presentation of the Implementation

The presentation of a systematic top-down design with accompanying correctness proofs is only useful after the design objectives have been completely understood. Therefore we will now introduce an explicit solution of the informal design task stated above. The systematic top-down design and verification according to the framework for the development of dependable systems introduced in Chapter 2 will be described in the subsequent sections.

The detailed design of the server system is shown in Figure 3.2. Before introducing the components of SYS, let us introduce the alphabets of the processes involved:

$$\alpha(CLIENT) = \{\!|\ app\_tx, app\_rc\ |\!\}$$

$$\begin{aligned}
\alpha(ABFTX) &= \{\!|\ app_tx, a_1\ |\!\} \\
\alpha(ABFRC) &= \{\!|\ app_rc, b_0, b_1\ |\!\}
\end{aligned}$$

$$\begin{aligned}
\alpha(NET0) &= \{\!|\ a_1, a_{02}, b_{02}, c_0, c_1, \mathit{off}_0, on_0\ |\!\} \\
\alpha(NET1) &= \{\!|\ a_1, a_{12}, b_{12}, c_0, c_1, \mathit{off}_1, on_1\ |\!\}
\end{aligned}$$

$$\begin{aligned}
\alpha(APP0) &= \{\!|\ a_{02}, b_{02}, d_0, d_1, \mathit{off}_0, on_0\ |\!\} \\
\alpha(APP1) &= \{\!|\ a_{12}, b_{12}, d_0, d_1, \mathit{off}_1, on_1\ |\!\}
\end{aligned}$$

$$\begin{aligned}
\alpha(CP0) &= \alpha(APP0) \cup \alpha(NET0) \\
\alpha(CP1) &= \alpha(APP1) \cup \alpha(NET1)
\end{aligned}$$

$$\alpha(DCP) = \alpha(CP0) \cup \alpha(CP1)$$

$$\alpha(SYS) = \alpha(ABFTX) \cup \alpha(ABFRC) \cup \alpha(DCP)$$

versions run on both CPUs of the fail-stop component. The main advantage of this technique is that it does not require to use complex *voting protocols*, because it may be safely assumed that the two CPUs never produce the identical error in the same cycle. Ideally, fail-stop hardware is combined with formally verified software.

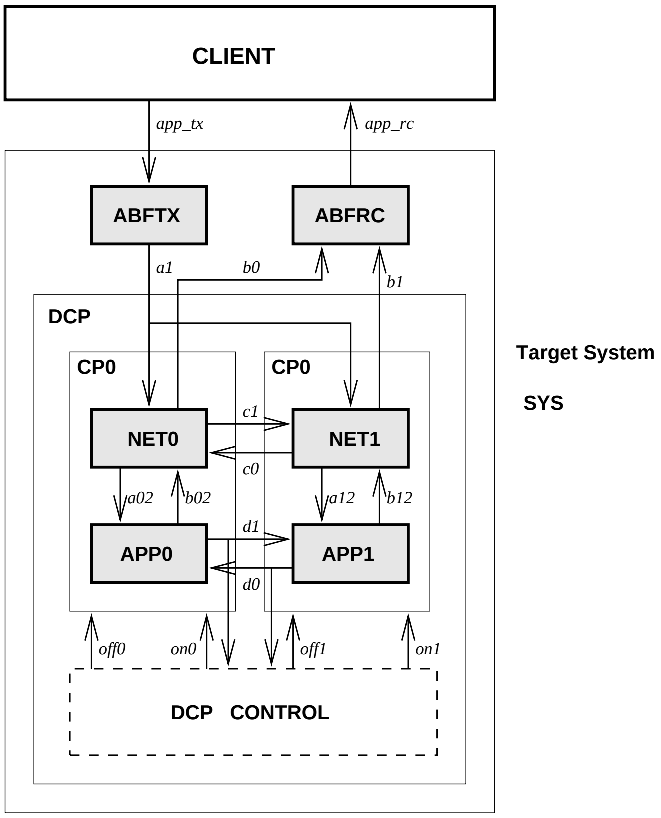

Figure 3.2: Full architecture of the fault-tolerant server system.

In these definitions, the channel alphabets are

Channel	Channel Alphabet
app_tx, app_rc	$DATA$
a_1, b_0, b_1	$BIT \times DATA$
$a_{02}, b_{02}, a_{12}, b_{12}$	$DATA$
c_0, c_1, d_0, d_1	$SIGNAL$

$DATA$ denotes the set of application data to be exchanged between $CLIENT$ and SYS. Define $BIT = \{0, 1\}$ to identify the application data on channels a_1, b_0, b_1. $SIGNAL = \{\varepsilon\}$ consists of a single event only, to be used for the generation of synchronisation messages.

On sub-system level[3], the server system SYS is decomposed into the dual computer system DCP and two filter components $ABFTX$ and $ABFRC$.

$$SYS = (ABFTX \parallel DCP \parallel ABFRC)$$

In the detailed design of DCP introduced below it will become apparent that in the case of the failure of one computer a duplicated message might be produced. To cope with such situations the process $ABFTX$ acts as an M-place buffer and associates an alternating bit

[3]See Section 2.3.2 for the terminology related to system decomposition.

with each new input from the *CLIENT*, before passing the pair (x, bit) of channel a_1 to the dual computer system. The representation of *ABFTX* as an explicit CSP process is

$$
\begin{aligned}
ABFTX &= TX(\langle\,\rangle, 0) \\[1em]
TX(s, bit) &= (\#s < M)\&app\_tx?x \to TX(s \frown \langle (x, bit) \rangle, 1 - bit) \\
&\quad \square \\
&\quad (\#s > 0)\&a_1!head(s) \to TX(tail(s), bit)
\end{aligned}
$$

DCP delivers each computation $f(x)$ together with the corresponding input bit on channels b_0 or b_1. Evaluating the bits, a duplicated message is recognised by the filter process *ABFRC* and discarded. The data part of valid messages is passed on to the client via channel *app_rc*:

$$
\begin{aligned}
ABFRC &= RC(0) \\[1em]
RC(bit) &= b_0?(y, b) \to D(bit, y, b) \\
&\quad \square \\
&\quad b_1?(y, b) \to D(bit, y, b) \\[1em]
D(bit, y, b) &= \textbf{if } bit = b \\
&\quad \textbf{then } app\_rc!y \to RC(1 - bit) \\
&\quad \textbf{else } RC(bit)
\end{aligned}
$$

It is assumed that *ABFTX* and *ABFRC* can be implemented on the client computer. Therefore these filter processes will fail if and only if *CLIENT* fails. As a consequence, we may simply assume that they always show normal behaviour.

On segment level, the dual-computer system *DCP* is structured as

$$DCP = (CP_0 \,\|\, CP_1 \,\|\, DCP\_CONTROL)$$

The segments CP_0 and CP_1 denote the two fail-stop computers. Before explaining their structure, we will discuss how to model the failure events in an appropriate way. An advantage of using fail-stop components is that we do not have to distinguish between a regular shutdown of a component and an unanticipated failure: In both cases the result will be that the component ceases to produce any visible events at its output interfaces. As a consequence we can model the crash situation just as a switch-off command given by an operator. The activities of an operator are external to CP_0 and CP_1, therefore they are modelled as an auxiliary process *DCP_CONTROL* to be described in detail below, sending on_i- and off_i-commands to the computers, where off_i now stands both for the regular switch-off and for the unexpected failure.

The basic design concept for the cooperation between CP_0 and CP_1 is a modified *hot standby* relationship with *master* and *standby* computer. 'Hot standby' denotes that both computers are active in normal operation, but only the master returns data to the client. In standard hot standby design, both master and standby process *everything* in parallel. This has the disadvantage that the redundancy does not increase the performance of the computer system *(static redundancy)* [7]. In our modified hot-standby design, the slave will cooperate with the master only to an extent which is absolutely necessary for not losing any jobs in case of a failure. Therefore some "spare CPU time" can be used to process non-critical jobs on the standby computer *(dynamic redundancy)*. These are simply interrupted, as soon as the master fails. The modified hot standby concept needs more time for recovery than the standard concept, because after a master failure certain jobs have to be re-started from the beginning, but it will turn out during the top-down design and the associated verification that the time required for recovery is globally bounded. Therefore our modified hot standby design is also suitable for hard real-time applications, if the client can accept this upper bound as the maximal response time.

The implementation of the modified hot standby concept will now be explained in detail. A message on a_1 is broadcasted to both CP_0 and CP_1, so that each job submitted by $CLIENT$ will be received by both fail-stop computers. The results are either delivered by CP_0 on channel b_0 or by CP_1 on channel b_1. The two computers exchange control signals on the channels c_0, c_1, d_0, d_1.

Each computer CP_0, CP_1 is decomposed into two software configuration items,

$$CP_i = NET_i \,\|\, APP_i$$

where NET_i denotes an internal network layer and APP_i an application layer, including the mechanisms needed for control of the processes P_i performing the computations requested by the client.

Each network layer NET_i is defined as

$$\begin{aligned} NET_i \; &= a_1?(x,b) \rightarrow NET_i \\ &\quad \Box \\ &\quad c_i \rightarrow NET_i \\ &\quad \Box \\ &\quad on_i \rightarrow (NET_{i1} \;\Box\; c_i \rightarrow NET_{i1}) \end{aligned}$$

$$NET_{i1} = NET_{i2} \,\hat{}\, (\mathit{off}_i \rightarrow NET_i)$$

$$\begin{aligned} NET_{i2} = a_1?(x,b) \rightarrow (&c_i \rightarrow NET_{i2} \\ &\Box \\ &a_{i2}!x \rightarrow b_{i2}?y \rightarrow b_i!(y,b) \rightarrow c_{(1-i)} \rightarrow NET_{i2}) \end{aligned}$$

The process state NET_i models the situation where CP_i has been switched off. In this mode inputs cannot be refused (otherwise the crash of CP_i might also block $CP_{(1-i)}$!), but only an on_i-event leads to further activities, described by NET_{i2}. Now each input $a_1.(x,b)$ is processed as follows: If CP_i acts as master, the data component x is passed via channel a_{i2} to the application layer. The result y received from APP_i on channel b_{i2} is again combined with the bit value b of the associated job and the pair (y,b) is sent to the environment on b_i. Afterwards CP_i sends a control signal on $c_{(1-i)}$ to $CP_{(1-i)}$ indicating that the job has been successfully processed. The slave computer $CP_{(1-i)}$ only keeps the input message $a_1.(x,b)$ until reception of the $c_{(1-i)}$-event. In normal operation, the communication on channel $a_{(1-i)2}$ is blocked. The operation of NET_{i2} can be interrupted by an off_i-event at any time, and the passive state NET_i is resumed. In this case the application layer $APP_{(1-i)}$ of the former slave will accept $a_{(1-i)2}$-inputs and start processing. As a consequence, a job that has been received by both computers but not yet acknowledged on $c_{(1-i)}$ will be automatically re-started by $CP_{(1-i)}$ and afterwards transmitted on $b_{(1-i)}$. This is the situation when duplicated messages might be produced: If the master CP_i fails *after* a job $(f(x), b)$ has been delivered on b_i but *before* the $c_{(1-i)}$-signal has been produced, $CP_{(1-i)}$ will re-process this job and produce the superfluous message $(f(x), b)$ on channel $b_{(1-i)}$. However, since this message carries the same bit as the previous one delivered by CP_i, it will be discarded by the filter process $ABFRC$.

The application layer APP_i is structured as

$$\begin{aligned} APP_i &= d_i \to on_i \to (d_{(1-i)} \to P_i \,\Box\, \mathit{off}_i \to APP_i) \\ &\quad \Box \\ &\quad on_i \to (d_{(1-i)} \to P_i \,\Box\, \mathit{off}_i \to APP_i) \\ P_i &= P_{i1} \hat{}\, (\mathit{off}_i \to APP_i) \\ P_{i1} &= a_{i2}?x \to b_{i2}!f(x) \to P_{i1} \end{aligned}$$

The data processing is performed in P_{i1}, the other events serve to control the master-standby relationship. Process state APP_i corresponds to NET_i, where the computer CP_i is passive. If CP_i receives an event d_i before the on_i-event has been issued, it acts as slave, observing the master by means of a *watchdog mechanism*, modelled by *watchdog event* event $d_{(1-i)}$: As long as the master is active, $d_{(1-i)}$ is blocked, so CP_i cannot activate its data processing component P_i. A failure of the master (modelled in the application layer by $\mathit{off}_{(1-i)}$ interrupting $P_{(1-i)1}$) leads to acceptance of $d_{(1-i)}$, and P_{i1} will start processing inputs on channel $a_{i2}?x$.

As indicated above, the admissible sequences of on_i, off_i, d_i-events are specified by the auxiliary component $DCP\_CONTROL$. With the techniques introduced in Chapter 2, we could introduce behavioural specifications $S_{DCP\_CONTROL}(s)$ and $\Delta_{DCP\_CONTROL}(s, s')$ describing the normal behaviour and the acceptable deviations from normal behaviour, respectively. We will, however, introduce an alternative technique which is suitable for verification via model checking: The normal behaviour informally described in Section 3.3.1 corresponds to a behaviour of $DCP\_CONTROL$, where only the events on_0, on_1 are produced and $\mathit{off}_0, \mathit{off}_1$ are forever refused. Acceptable behaviour is modelled by a behaviour of $DCP\_CONTROL$ allowing off_i-events to occur in certain situations covered by the fault hypotheses.

Setting

$$\alpha(DCP\_CONTROL) = \{| \; a_1, b_0, b_1, on_0, on_1, \mathit{off}_0, \mathit{off}_1, d_0, d_1 \; |\}$$

$$DCP\_CONTROL \quad = NORMAL \sqcap ACCEPTABLE$$

the two types of behaviour can be modelled explicitly as CSP processes. Since DCP runs in parallel with $DCP\_CONTROL$ and synchronises over the events a_1, b_0, b_1, on_0, on_1, off_0, off_1, d_0, d_1, only those sequences of failure events, watchdog signals, switch-on events and application messages can occur that are accepted by $DCP\_CONTROL$. Observe that it is not our objective to *implement* $DCP\_CONTROL$. Instead, the process models our *hypotheses* about the possible sequences of events that may occur. Our goal is to verify that SYS meets its normal behaviour and acceptable behaviour requirements, as long as the hypotheses expressed by $NORMAL$ and $ACCEPTABLE$ are valid.

We define

$$\begin{aligned} NORMAL = {} & on_0 \to d_1 \to on_1 \to RUN_{\{\| a_1, b_0, b_1 \|\}} \\ & \sqcap \\ & on_1 \to d_0 \to on_0 \to RUN_{\{\| a_1, b_0, b_1 \|\}} \end{aligned}$$

This describes the behaviour where either CP_0 or CP_1 is switched on first to act as the master, then the standby computer is activated, and afterwards the system is never switched off, i. e., it will never fail. Observe that the switch-on procedure requires that the second computer CP_i that is to act as standby will only be activated *after* the watchdog signal d_i has been produced. Otherwise a deadlock could occur in processes APP_1, APP_2, if both computers tried to force each other to act as standby.

The acceptable deviations from normal behaviour are modelled by process $ACCEPTABLE$, which is in turn decomposed into three parallel components, each expressing one aspect of the fault hypotheses.

$$ACCEPTABLE = (FH_1 \parallel (on_0 \to on_1 \to FH_2 \,\square\, on_1 \to on_0 \to FH_2) \parallel FH_3)$$

$$\begin{aligned} \alpha(FH_1) &= \{on_0, on_1, \mathit{off}_0, \mathit{off}_1, d_0, d_1\} \\ \alpha(FH_2) &= \{\| \; on_0, on_1, \mathit{off}_0, \mathit{off}_1, a_1, b_0, b_1 \; \|\} \\ \alpha(FH_3) &= \{\| \; \mathit{off}_0, \mathit{off}_1, a_1 \; \|\} \end{aligned}$$

The fault hypothesis modelled by process FH_1 describes the appropriate causal relationships between failure events and the re-integration of repaired components. Furthermore it controls the master-standby relationship between both computers, which depends on the sequence of failure events and re-integration.

$$\begin{aligned} FH_1 \quad &= on_0 \to d_1 \to on_1 \to FHMS \sqcap on_1 \to d_0 \to on_0 \to FHSM \\ FHMS &= \mathit{off}_0 \to d_0 \to on_0 \to FHSM \sqcap \mathit{off}_1 \to on_1 \to FHMS \\ FHSM &= \mathit{off}_0 \to on_0 \to FHSM \sqcap \mathit{off}_1 \to d_1 \to on_1 \to FHMS \end{aligned}$$

FH_1 corresponds to the system initialisation performed by $NORMAL$. If CP_0 is switched on first, the following acceptable behaviour is modelled by $FHMS$: CP_0 acts as master, CP_1 as standby. If CP_0 fails (is switched off) in state $FHMS$ while acting as master, the switch-on on_0 must not occur before CP_1 has detected the failure and issued the watchdog signal d_0 forcing CP_0 to act as standby after re-integration. Further control is then performed by $FHSM$, describing the situation when CP_1 acts as master. Furthermore, $FHMS$ expresses that a second failure must not occur while one computer is still being repaired. Finally, $FHMS$ specifies that after a failure of the standby computer CP_1 re-integration can take place immediately, because the watchdog signal d_1 has already been issued before, and the master-standby relationship is not changed after a standby failure. $FHSM$ describes the analogous situations for CP_1 acting as master.

Process FH_2 is an extension of FH_1, specifying the relationship between application messages and failure events. If both computers are active, a failure may happen at any time without any jobs being lost. The situation is different, as soon as one computer has already failed. If the master has started a job *before* re-integration takes place and the job is still active *after* switch-on of the repaired component, the full reliability is only restored when this job has been completed and delivered by the master. Formally speaking, our fault hypotheses for acceptable behaviour require that sequences

$$\ldots \rightarrow \mathit{off}_i \rightarrow a_1.(x,b) \rightarrow on_i \rightarrow \mathit{off}_{(1-i)}$$

will never occur. On the other hand, another failure of the re-integrated component can be tolerated at any time, as long as the master continues its operation. These requirements lead to the following lengthy, but simple structure for FH_2:

$$FH_2 \;= x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{21} \;\square\; \mathit{off}_0 \rightarrow FH_{22} \;\square\; \mathit{off}_1 \rightarrow FH_{23} \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_2$$

$$FH_{21} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{21} \;\square\; \mathit{off}_0 \rightarrow FH_{24} \;\square\; \mathit{off}_1 \rightarrow FH_{25} \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_2$$

$$FH_{22} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{24} \;\square\; on_0 \rightarrow FH_2 \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_{22}$$

$$FH_{23} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{25} \;\square\; on_1 \rightarrow FH_2 \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_{23}$$

$$FH_{24} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{24} \;\square\; on_0 \rightarrow FH_{26} \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_{22}$$

$$FH_{25} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{25} \;\square\; on_1 \rightarrow FH_{27} \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_{23}$$

$$FH_{26} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{26} \;\square\; \mathit{off}_0 \rightarrow FH_{24} \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_2$$

$$FH_{27} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_{27} \;\square\; \mathit{off}_1 \rightarrow FH_{25} \;\square\; x : \{\!|\; b_0, b_1 \;|\!\} \rightarrow FH_2$$

In process states $FH_{22}, \ldots, FH_{25}$ only one computer is active, so a second failure must not occur. States FH_{24}, FH_{25} model the situation when a job is processed while only one computer is active. If switch-on of the repaired component takes place during that time, a state transition into FH_{26} or FH_{27} takes place, so that only a new failure of the repaired component is accepted, until the pending job has been delivered on b_0 or b_1.

Finally, FH_3 expresses that the frequency of failures and corresponding re-integration is low enough to admit at least one application input between two crashes.

$$FH_3 \;= x : \{\mathit{off}_0, \mathit{off}_1\} \rightarrow FH_{31} \;\square\; x : \{\!|\; a_1 \;|\!\} \rightarrow FH_3$$

$$FH_{31} = x : \{\!|\; a_1 \;|\!\} \rightarrow FH_3$$

3.3.3 System Requirements

We will now present the server system *SYS* by means of a systematic top-down decomposition, following the concept introduced in Chapter 2.

Operational Environment

The operational environment is given by process $CLIENT$ operating in parallel with the server system SYS and not restricted by any boundary conditions.

$$\mathcal{E}(SYS) = (CLIENT \parallel SYS)$$

$$CLIENT \ \mathbf{sat} \ true$$

Application Requirements

The server system should show the normal behaviour $S_{SYS}(s, R)$, which formalises the requirements described in Section 3.3.2.

$$\alpha(SYS \setminus L_{SYS}) = \{\!|\ app\_tx, app\_rc\ |\!\}$$

$$\mathcal{E}(SYS) \setminus L_{SYS} \ \mathbf{sat} \ S_{SYS}(s, R)$$

$$S_{SYS}(s, R) \equiv_{df} SAFETRACE(s) \wedge NOBLOCK(s, R)$$

$$SAFETRACE(s) \equiv_{df} val^*(s \upharpoonright \{\!|\ app\_rc\ |\!\}) \leq^N f^*(val^*(s \upharpoonright \{\!|\ app\_tx\ |\!\}))$$

$$\begin{aligned} &NOBLOCK(s, R) \equiv_{df} \\ &\qquad (\#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) = \#(s \upharpoonright \{\!|\ app\_tx\ |\!\}) \wedge R \cap \{\!|\ app\_tx\ |\!\} = \emptyset \\ &\qquad\qquad \vee\ \#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) < \#(s \upharpoonright \{\!|\ app\_tx\ |\!\}) \wedge \{\!|\ app\_rc\ |\!\} \not\subseteq R) \end{aligned}$$

Note that in contrast to Chapter 2, we use behavioural specifications about traces and refusals, because also the proofs about deadlock freedom on sub-system level will be carried out using the laws of the failures-divergence model.

The first conjunct $SAFETRACE(s)$ in specification $S_{SYS}(s, R)$ contains the safety properties about traces of $SYS \setminus L_{SYS}$. It states that the outputs on channel app_rc are the results of the computation $f(x)$ applied to the inputs received on channel app_tx. The level of buffering, as far as visible on this interface is globally bounded by some $N > 0$. The second conjunct $NOBLOCK(s, R)$ summarises the requirements with respect to deadlock freedom: If SYS has processed every input $(\#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) = \#(s \upharpoonright \{\!|\ app\_tx\ |\!\})$, new inputs on app_tx cannot be refused. Conversely, as long as some outputs have not yet been delivered, communication on app_rc cannot be blocked by SYS.

External Threat Analysis

As indicated in Section 3.3.2, external threats are not considered in this case study, the only fault hypotheses being concerned with the dual computer system. Since DCP is a sub-system of the complete server system SYS, these are regarded as internal threats.

3.3.4 System Architecture

System Structure

On sub-system level, the detailed design depicted in Figure 3.2 looks as shown in Figure 3.3. The meaning of the sub-systems $ABFTX$, DCP and $ABFRC$ has already been described in Section 3.3.2.

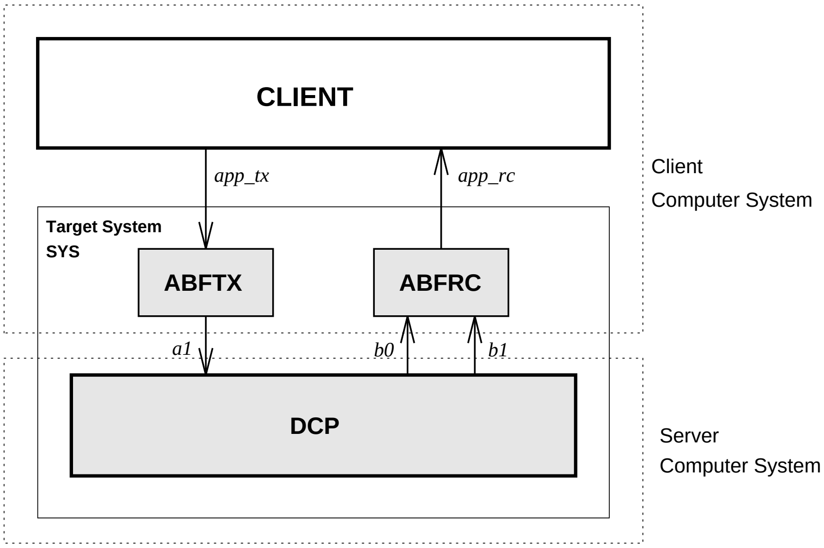

Figure 3.3: Server system architecture.

$$
\begin{aligned}
\alpha(SYS \setminus L_{SYS}) &= \{\!| \; app\_tx, app\_rc \; |\!\} \\
L_{SYS} &= \{\!| \; a_1, b_0, b_1 \; |\!\} \cup L_{DCP} \\
\alpha(DCP) &= \{\!| \; a_1, b_0, b_1 \; |\!\} \cup L_{DCP} \\
\alpha(ABFTX) &= \{\!| \; app\_tx, a_1 \; |\!\} \\
\alpha(ABFRC) &= \{\!| \; app\_rc, b_0, b_1 \; |\!\} \\
\\
SYS &= \mathcal{A}_{SYS}(ABFTX, DCP, ABFRC) \\
&= (ABFTX \parallel DCP \parallel ABFRC)
\end{aligned}
$$

Component Specifications

The implicit normal behaviour specifications of the sub-systems look as follows:

$$ABFTX \textbf{ sat } S_{TX}(s, R)$$

$$
\begin{aligned}
& S_{TX}(s, R) \equiv_{df} \\
& \quad data^*(s \upharpoonright \{\!| \; a_1 \; |\!\}) \leq^M val^*(s \upharpoonright \{\!| \; app\_tx \; |\!\}) \\
& \quad \wedge \\
& \quad \rho(s \upharpoonright \{\!| \; a_1 \; |\!\}) = s \upharpoonright \{\!| \; a_1 \; |\!\} \\
& \quad \wedge \\
& \quad (\#(s \upharpoonright \{\!| \; a_1 \; |\!\}) = \#(s \upharpoonright \{\!| \; app\_tx \; |\!\}) \wedge R \cap \{\!| \; app\_tx \; |\!\} = \emptyset \\
& \qquad\qquad \vee \#(s \upharpoonright \{\!| \; a_1 \; |\!\}) < \#(s \upharpoonright \{\!| \; app\_tx \; |\!\}) \wedge \{\!| \; a_1 \; |\!\} \not\subseteq R)
\end{aligned}
$$

Here we have re-used the functions $\rho, data(x), bit(x)$ introduced in Chapter 2: ρ deletes events carrying duplicated bits from traces on channels with alphabet $DATA \times BIT$, $data$ retrieves the data component x from events $c.(x, b)$ on such channels and bit retrieves the bit component b.

According to $S_{TX}(s, R)$, process $ABFTX$ copies the inputs received on app_tx to the data component of channel a_1, thereby associating bit values such that the filter ρ would not discard any a_1-event, i. e., the bits associated by $ABFTX$ strictly alternate.

The receive component of the protocol is defined as

$$ABFRC \ \mathbf{sat} \ S_{RC}(s, R)$$

$$\begin{array}{l} S_{RC}(s, R) \equiv_{df} \\ \quad val^*(s \upharpoonright \{\!|\ app\_rc \ |\!\}) \leq^1 data^*(\rho(s \upharpoonright \{\!|\ b_0, b_1 \ |\!\})) \\ \quad \wedge \\ \quad (\#(s \upharpoonright \{\!|\ app\_rc \ |\!\}) = \#\rho(s \upharpoonright \{\!|\ b_0, b_1 \ |\!\}) \wedge R \cap \{\!|\ b_0, b_1 \ |\!\} = \emptyset \\ \qquad\qquad\qquad \vee \ \#(s \upharpoonright \{\!|\ app\_rc \ |\!\}) < \#\rho(s \upharpoonright \{\!|\ b_0, b_1 \ |\!\}) \wedge \{\!|\ app\_rc \ |\!\} \not\subseteq R) \end{array}$$

As implied by the definition of ρ, $ABFRC$ discards duplicated consecutive messages, identified by their identical bits.

The normal behaviour of the dual computer system DCP on interface level is

$$DCP \setminus L_{DCP} \ \mathbf{sat} \ S_{DCP}(s, R)$$

$$\begin{array}{l} S_{DCP}(s, R) \equiv_{df} \\ \quad s \upharpoonright \{\!|\ b_1 \ |\!\} = \langle \ \rangle \\ \quad \wedge \\ \quad val^*(s \upharpoonright \{\!|\ b_0 \ |\!\}) \leq^1 ((f \circ data) \times bit)^*(s \upharpoonright \{\!|\ a_1 \ |\!\}) \\ \quad \wedge \\ \quad (\#(s \upharpoonright \{\!|\ b_0 \ |\!\}) = \#(s \upharpoonright \{\!|\ a_1 \ |\!\}) \wedge R \cap \{\!|\ a_1 \ |\!\} = \emptyset \\ \qquad\qquad\qquad \vee \ \#(s \upharpoonright \{\!|\ b_0 \ |\!\}) < \#(s \upharpoonright \{\!|\ a_1 \ |\!\}) \wedge \{\!|\ b_0 \ |\!\} \not\subseteq R) \end{array}$$

This reflects the situation where CP_0 acts as master. Since the master-standby relationship is never changed in normal behaviour, DCP will always deliver results on channel b_0. In these results, f has been applied to the data component of a_1 and the bit component of a_1 is again attached to the return message on b_0.

Internal Threat Analysis

Our system design allocates $ABFTX$ and $ABFRC$ on the client computer. Therefore we are not interested in the impact of their possible failure, since in such a case the client will not have a chance to operate anyway. As a consequence it can be assumed that for these protocol processes normal behaviour is guaranteed.

On sub-system level, the deviation of $DCP \setminus L_{DCP}$ from normal behaviour can be specified as follows:

$$
\begin{aligned}
&\Delta_{DCP}(s, s', R, R') \equiv_{df} \\
&\quad s' \upharpoonright \{\!|\ a_1\ |\!\} = s \upharpoonright \{\!|\ a_1\ |\!\} \\
&\quad \wedge \\
&\quad val^*(\rho(s' \upharpoonright \{\!|\ b_0, b_1\ |\!\})) = val^*(s \upharpoonright \{\!|\ b_0\ |\!\}) \\
&\quad \wedge \\
&\quad (\{last(s'), last(front(s'))\} \subseteq \{\!|\ b_0, b_1\ |\!\} \Rightarrow \\
&\qquad\qquad last(front(front(s'))) \in \{\!|\ a_1\ |\!\} \wedge R' \cap \{\!|\ a_1\ |\!\} = \emptyset) \\
&\quad \wedge \\
&\quad (last(s') \in \{\!|\ b_0, b_1\ |\!\} \Rightarrow \{\!|\ a_1, b_0, b_1\ |\!\} \not\subseteq R') \\
&\quad \wedge \\
&\quad (\#\rho(s' \upharpoonright \{\!|\ b_0, b_1\ |\!\}) < \#(s \upharpoonright \{\!|\ b_0\ |\!\}) \Rightarrow \{\!|\ b_0, b_1\ |\!\} \not\subseteq R')
\end{aligned}
$$

The inputs accepted on a_1 are the same as when showing normal behaviour. Outputs now have to be expected from both channels b_0 and b_1. However, it can at least be guaranteed that the sequence of b_0, b_1-events filtered by ρ carries the same values as channel b_0 in the normal behaviour situation. The conjuncts involving refusals state that not more than two consecutive b_0, b_1-events may happen, and after these events another a_1-input cannot be refused. After one b_i-event it is uncertain whether an a_1-input will be accepted, but at least the system will not block completely. If the filtered output sequence does not yet contain the full sequence of processed inputs, $DCP \setminus L_{DCP}$ will always be ready to deliver another output.

Applying the threat introduction operator $\wr$ introduced in Chapter 2 and the consequence rule, we derive a specification of $DCP \setminus L_{DCP}$ in presence of threats as

$$((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \ \mathbf{sat} \ (S_{DCP} \wr \Delta_{DCP})(s, R)$$

$$
\begin{aligned}
&(S_{DCP} \wr \Delta_{DCP})(s, R) \equiv_{df} \\
&\quad \rho(s \upharpoonright \{\!|\ a_1\ |\!\}) = s \upharpoonright \{\!|\ a_1\ |\!\} \Rightarrow \\
&\qquad (val^*(\rho(s \upharpoonright \{\!|\ b_0, b_1\ |\!\})) \leq^1 ((f \circ data) \times bit)^*(s \upharpoonright \{\!|\ a_1\ |\!\}) \\
&\qquad \wedge \\
&\qquad (\{last(s), last(front(s))\} \subseteq \{\!|\ b_0, b_1\ |\!\} \Rightarrow \\
&\qquad\qquad\qquad last(front(front(s))) \in \{\!|\ a_1\ |\!\} \wedge R \cap \{\!|\ a_1\ |\!\} = \emptyset) \\
&\qquad \wedge \\
&\qquad (last(s) \in \{\!|\ b_0, b_1\ |\!\} \Rightarrow \{\!|\ a_1, b_0, b_1\ |\!\} \not\subseteq R) \\
&\qquad \wedge \\
&\qquad (\#\rho(s \upharpoonright \{\!|\ b_0, b_1\ |\!\}) < \#(s \upharpoonright \{\!|\ a_1\ |\!\}) \Rightarrow \{\!|\ b_0, b_1\ |\!\} \not\subseteq R))
\end{aligned}
$$

3.3.5 Verification on System Level

We will now perform the verification steps required according to Chapter 2, in order to prove that the normal and acceptable behaviour specifications of the sub-systems imply the behavioural requirements for the full system SYS. Since acceptable behaviour is identical to normal behaviour on system level, it has to be shown that both normal and acceptable behaviour of the sub-systems implies specification $S_{SYS(s,R)}$.

In contrast to the techniques applied in Chapter 2 we will now verify the safety properties as well as deadlock freedom in one step, using the proof techniques provided by the failures-divergence model of CSP.

Normal Behaviour Verification

Theorem 1 (System Verification Obligation – Normal Behaviour) *With the above definitions of system architecture and normal behaviour specifications*

$$\frac{\begin{array}{l} ABFTX \ \mathbf{sat} \ S_{TX}(s,R) \\ ABFRC \ \mathbf{sat} \ S_{RC}(s,R) \\ DCP \setminus L_{DCP} \ \mathbf{sat} \ S_{DCP}(s,R) \end{array}}{(ABFTX \parallel DCP \parallel ABFRC) \setminus \alpha(DCP) \ \mathbf{sat} \ S_{SYS}(s,R)}$$

holds.
□

In this theorem observe that due to the system structure introduced for SYS in 3.3.4 the expression $(ABFTX \parallel DCP \parallel ABFRC) \setminus \alpha(DCP)$ is equal to $SYS \setminus L_{SYS}$. We skip the proof, because it is an analogous version of the one presented for Theorem 2 below.

Exceptional Behaviour Verification

Theorem 2 (System Verification Obligation – Acceptable Behaviour) *With the above definitions of system architecture and exceptional behaviour specification of component* $DCP \setminus L_{DCP}$*,*

$$\frac{\begin{array}{l} ABFTX \ \mathbf{sat} \ S_{TX}(s,R) \\ ABFRC \ \mathbf{sat} \ S_{RC}(s,R) \\ ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \ \mathbf{sat} \ (S_{DCP} \wr \Delta_{DCP})(s,R) \end{array}}{(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC) \setminus \alpha(DCP) \ \mathbf{sat} \ S_{SYS}(s,R)}$$

holds.

Before presenting the proof of Theorem 2 we need two additional lemmas. The first one presents a stronger proof rule about the specifications of parallel processes than the one given in [44, p. 124]:

Lemma 6 *The law*

$$\frac{\begin{array}{l} P \ \mathbf{sat} \ S_P(s,R) \\ Q \ \mathbf{sat} \ S_Q(s,R) \end{array}}{(P \parallel Q) \ \mathbf{sat} \ (\exists\, X, Y \bullet R = X \cup Y \wedge S_P(s \upharpoonright \alpha(P), X) \wedge S_Q(s \upharpoonright \alpha(Q), Y))}$$

holds in the failures-divergence model provided that

$$Div(P) \cap \{s : Traces(P \parallel Q) \bullet s \upharpoonright \alpha(P)\} = \emptyset,$$
$$Div(Q) \cap \{s : Traces(P \parallel Q) \bullet s \upharpoonright \alpha(Q)\} = \emptyset$$

Proof.
From the semantics of $\|$ we know that [44, p. 128]

$$\begin{aligned}&\mathit{Div}(P \parallel Q) = \{s, t : (\alpha(P) \cup \alpha(Q))^* \mid \\ &\quad (s \restriction \alpha(P) \in \mathit{Div}(P) \wedge s \restriction \alpha(Q) \in \mathit{Traces}(Q) \\ &\qquad \vee\, s \restriction \alpha(P) \in \mathit{Traces}(P) \wedge s \restriction \alpha(Q) \in \mathit{Div}(Q)) \bullet s^\frown t\}\end{aligned}$$

Therefore the premises of the lemma imply $\mathit{Div}(P \parallel Q) = \emptyset$. As a consequence, the semantics of $\|$ implies [44, p. 131]

$$\begin{aligned}&\mathit{Fail}(P \parallel Q) = \{s : (\alpha(P) \cup \alpha(Q))^*;\ X : \mathbb{P}\,\alpha(P);\ Y : \mathbb{P}\,\alpha(Q) \mid \\ &\quad (s \restriction \alpha(P), X) \in \mathit{Fail}(P) \wedge (s \restriction \alpha(Q), Y) \in \mathit{Fail}(Q) \bullet (s, X \cup Y)\}\end{aligned}$$

so every failure (s, R) of $(P \parallel Q)$ has a refusal of the form $R = X \cup Y$. Since according to the premises of the lemma every failure of P satisfies S_P and every failure of Q satisfies S_Q, both $S_P(s \restriction \alpha(P), X)$ and $S_Q(s \restriction \alpha(Q), Y)$ hold.
□

The second lemma to be used in the proof of Theorem 2 states that if a process does not diverge after application of the hiding operator, the number of consecutive hidden events it may perform must be bound by a function depending only the trace of visible events:

Lemma 7 *For CSP process P and $H \subseteq \alpha(P)$ define*

$$\begin{aligned}&\mathit{NODIV}(H) \equiv_{df} \\ &\quad (\exists\, \beta : (\alpha(P) - H)^* \to \mathbb{N} \bullet \forall\, s : \mathit{Traces}(P) \bullet \#(s \restriction H) \leq \beta(s \restriction (\alpha(P) - H)))\end{aligned}$$

Then $\mathit{Div}(P \setminus H) = \emptyset$ implies P **sat** *$\mathit{NODIV}(H)$.*

Proof.
Suppose that P does not satisfy $\mathit{NODIV}(H)$, so that the associated bounding function does not exist. We will show that this implies divergence of $P \setminus H$. Let $n \in \mathbb{N}$. Define $\beta_n : (\alpha(P) - H)^* \to \mathbb{N}$ by $\beta_n(v) = n \cdot (1 + \#v)$. According to our assumption there exists a trace u of P such that $\beta_n(u \restriction (\alpha(P) - H)) = n \cdot (1 + \#(u \restriction (\alpha(P) - H))) < \#(u \restriction H)$. Then u contains at least one section of more than n consecutive H-events without any interleaving events of $(\alpha(P) - H)$. Since this construction is independent on n, there exists a mapping $\gamma : \mathbb{N} \to \mathit{Traces}(P)$ such that $\gamma(n) = v^\frown w$ and w consists of at least n consecutive H-events and v does *not* contain n consecutive H-events. The range of γ is a subtree of $\mathit{Traces}(P)$ and therefore finitely branching. It is infinite since $\#(\gamma(n)) \geq n$ and γ is a total function on $\mathbb{N}$. Therefore application of König's Lemma [71, p. 118] yields the existence of an infinite trace u_0 in ran γ. By construction of γ, this trace must contain *for each* $n \in \mathbb{N}$ a section of at least n consecutive H-events, because otherwise u_0 would be finite. Now the semantics of the hiding operator implies that $P \setminus H$ diverges [44, p. 128], and this proves the lemma.
□

Proof of Theorem 2.

Our proof is structured as follows: Proof obligations 1 to 5 establish that the premises of the theorem imply various aspects of non-divergence for $ABFTX$, $ABFRC$ and

$((DCP \setminus L_{DCP}) \wr \Delta_{DCP})$. This is used in proof obligation 6 to derive an auxiliary specification satisfied by $(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}$. This specification is suitable to derive

$$\begin{aligned}&(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\} \\ &\qquad\qquad\qquad \textbf{sat}\ SAFETRACE(s) \wedge NOBLOCK(s, R)\end{aligned}$$

using the consequence rule and conjunction rule. This derivation is performed in two steps, the first proving trace safety $SAFETRACE(s)$, the second deadlock freedom $NOBLOCK(s, R)$. Since $S_{SYS}(s, R)$ is equivalent to the conjunction of $SAFETRACE(s)$ and $NOBLOCK(s, R)$ (see page 67), this establishes the theorem.

Proof Obligation 1. If $ABFTX$ **sat** $S_{TX}(s, R)$ then $Div(ABFTX) = \emptyset$.

Suppose s is a trace of $ABFTX$. In case $\#(s \restriction \{\!|\ a_1\ |\!\}) = \#(s \restriction \{\!|\ app\_tx\ |\!\})$ specification $S_{TX}(s, R)$ asserts that $\{\!|\ app\_tx\ |\!\}$ cannot be refused and in case $\#(s \restriction \{\!|\ a_1\ |\!\}) < \#(s \restriction \{\!|\ app\_tx\ |\!\})$ it asserts that at least one element of $\{\!|\ a_1\ |\!\}$ cannot be refused. Since $S_{TX}(s, R)$ also guarantees that $\#(s \restriction \{\!|\ a_1\ |\!\}) \leq \#(s \restriction \{\!|\ app\_tx\ |\!\})$, no additional cases are left. As a consequence $ABFTX$ cannot diverge after s, because otherwise $ABFTX/s$ could refuse the full alphabet $\alpha(ABFTX)$. This proves obligation 1.

Proof Obligation 2. If $ABFRC$ **sat** $S_{RC}(s, R)$ then $Div(ABFRC) = \emptyset$.

This is shown in analogy to obligation 1.

Proof Obligation 3. Under the premises of the theorem
$Div(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) = \emptyset$ holds.

From the validity of proof obligations 1 and 2 we know that $ABFTC$ and $ABFRC$ do not diverge. Inserting this information into the failures-divergence semantics of $\|$ ([44, p. 128]) yields

$$\begin{aligned}&Div(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) = \\ &\quad \{s, w : (\alpha(ABFTX) \cup \alpha(ABFRC) \cup \alpha((DCP \setminus L_{DCP}) \wr \Delta_{DCP}))^* \mid \\ &\qquad s \restriction \alpha(ABFTX) \in Traces(ABFTX) \wedge s \restriction \alpha(ABFTX) \in Traces(ABFTX) \wedge \\ &\qquad s \restriction \alpha((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \in Div((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \bullet s ^\frown w\}\end{aligned}$$

that is, only $((DCP \setminus L_{DCP}) \wr \Delta_{DCP})$ can contribute to a divergence of the three cooperating processes.

Now let $u \in Div(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC)$, such that $front(u) \notin Div(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC)$. Since $u \restriction \alpha(ABFTX)$ is a trace of $ABFTX$ and this process is assumed to satisfy S_{TX}, it follows that $\rho(u \restriction \{\!|\ a_1\ |\!\}) = u \restriction \{\!|\ a_1\ |\!\}$. As a consequence $u \restriction \alpha((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) = u \restriction \{\!|\ a_1, b_0, b_1\ |\!\}$ satisfies the premise of specification $(S_{DCP} \wr \Delta_{DCP})$, so

$$
\begin{array}{l}
val^*(\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\})) \leq^1 ((f \circ data) \times bit)^*(u \upharpoonright \{\!|\ a_1\ |\!\}) \\
\wedge \\
(\{last(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}), last(front(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}))\} \subseteq \{\!|\ b_0, b_1\ |\!\} \Rightarrow \\
\qquad\qquad last(front(front(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}))) \in \{\!|\ a_1\ |\!\} \wedge X \cap \{\!|\ a_1\ |\!\} = \emptyset) \\
\wedge \\
(last(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \in \{\!|\ b_0, b_1\ |\!\} \Rightarrow \{\!|\ a_1, b_0, b_1\ |\!\} \not\subseteq X) \\
\wedge \\
(\#\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\}) < \#(u \upharpoonright \{\!|\ a_1\ |\!\}) \Rightarrow \{\!|\ b_0, b_1\ |\!\} \not\subseteq X)
\end{array}
$$

holds, where X is an arbitrary refusal of $((DCP \setminus L_{DCP}) \wr \Delta_{DCP})/(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\})$. Specifically, this predicate holds for $X = \{\!|\ a_1, b_0, b_1\ |\!\}$, because $((DCP \setminus L_{DCP}) \wr \Delta_{DCP})$ diverges after $u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}$ and therefore the full alphabet is also a refusal. As a consequence none the premises of the last three conjuncts may be valid, because each consequence states that X is a proper subset of the alphabet $\{\!|\ a_1, b_0, b_1\ |\!\}$. Suppose therefore that $u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}$ neither satisfies $\{last(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}), last(front(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}))\} \subseteq \{\!|\ b_0, b_1\ |\!\}$ nor $last(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \in \{\!|\ b_0, b_1\ |\!\}$. Then $last(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \in \{\!|\ a_1\ |\!\}$. The first conjunct of the above predicate yields

$$\#(\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\})) \leq \#(u \upharpoonright \{\!|\ a_1\ |\!\})$$

As a consequence

$$
\begin{array}{ll}
\#(\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\})) & \\
& [last(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \in \{\!|\ a_1\ |\!\}] \\
\quad = \#(\rho(front(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \upharpoonright \{\!|\ b_0, b_1\ |\!\})) & \\
& [\text{above predicate also holds for } front(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\})] \\
\quad \leq \#(front(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \upharpoonright \{\!|\ a_1\ |\!\}) & \\
& [last(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \in \{\!|\ a_1\ |\!\}] \\
\quad < \#(u \upharpoonright \{\!|\ a_1\ |\!\}) &
\end{array}
$$

Therefore the premise of the last conjunct in the above predicate is valid, a contradiction. As a consequence, $X = \{\!|\ a_1, b_0, b_1\ |\!\}$ cannot be a refusal of $((DCP \setminus L_{DCP}) \wr \Delta_{DCP})/(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\})$ and therefore $((DCP \setminus L_{DCP}) \wr \Delta_{DCP})$ does not diverge after $u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}$. This proves obligation 3.

Proof Obligation 4. Under the premises of the theorem $(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)$ **sat** $NODIV(\{\!|\ a_1, b_0, b_1\ |\!\})$ holds.

From proof obligation 3 we know that $(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)$ does not diverge. Therefore the semantics of $\parallel$ ([44, p. 131]) implies

$$s \upharpoonright \alpha(ABFTX) = s \upharpoonright \{\!|\ app\_tx, a_1\ |\!\} \in Traces(ABFTX)$$

and

$$s \upharpoonright \alpha((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) = s \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\} \in Traces((DCP \setminus L_{DCP}) \wr \Delta_{DCP})$$

Since specification S_{TX} holds for $ABFTX$, this implies

$$\#(s \restriction \{\!|\ a_1\ |\!\}) \leq \#(s \restriction \{\!|\ app\_tx\ |\!\})$$

Moreover, the validity of S_{TX} implies $\rho(s \restriction \{\!|\ a_1\ |\!\}) = s \restriction \{\!|\ a_1\ |\!\}$, so the premise of specification $(S_{DCP} \wr \Delta_{DCP})$ is fulfilled. This results in

$$\{last(s \restriction \{\!|\ a_1, b_0, b_1\ |\!\}), last(front(s \restriction \{\!|\ a_1, b_0, b_1\ |\!\}))\} \subseteq \{\!|\ b_0, b_1\ |\!\} \Rightarrow \\ last(front(front(s \restriction \{\!|\ a_1, b_0, b_1\ |\!\}))) \in \{\!|\ a_1\ |\!\} \wedge R \cap \{\!|\ a_1\ |\!\} = \emptyset$$

As a consequence the estimate

$$\#(s \restriction \{\!|\ b_0, b_1\ |\!\}) \leq 2 \cdot \#(s \restriction \{\!|\ a_1\ |\!\}) \leq 2 \cdot \#(s \restriction \{\!|\ app\_tx\ |\!\})$$

and therefore

$$\#(s \restriction \{\!|\ a_1, b_0, b_1\ |\!\}) = \#(s \restriction \{\!|\ a_1\ |\!\}) + \#(s \restriction \{\!|\ b_0, b_1\ |\!\}) \leq 3 \cdot \#(s \restriction \{\!|\ app\_tx\ |\!\})$$

holds. This shows that $NODIV(\{\!|\ a_1, b_0, b_1\ |\!\})$ holds with bounding function $\beta(s|\{\!|\ app\_tx, app\_rc\ |\!\}) = 3 \cdot \#(s \restriction \{\!|\ app\_tx\ |\!\})$, and this proves obligation 4.

Proof Obligation 5. Under the premises of the theorem $Div((ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}) = \emptyset$ holds.

According to the semantics of $\setminus$ we have

$$Div(P \setminus H) = \\ \{s : \alpha(P)^*;\ t : (\alpha(P) - H)^* \mid s \in Div(P) \vee \\ (\forall n : \mathbb{N} \bullet \exists u : H^* \bullet n < \#u \wedge s^\frown u \in Traces(P)) \bullet (s \restriction (\alpha(P) - H))^\frown t\}$$

[44, p. 128][4]. Since proof obligations 3 and 4 hold, $(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)$ neither diverges nor engages into an unbounded number of consecutive $\{\!|\ a_1, b_0, b_1\ |\!\}$-events, so $Div(P \setminus H)$ is empty for $P = (ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)$ and $H = \{\!|\ a_1, b_0, b_1\ |\!\}$. This shows the validity of proof obligation 5.

Proof Obligation 6. Under the premises of the theorem

[4] Observe that this could be simply, but less intuitively, written as $Div(P \setminus H) = \{s : \alpha(P)^*;\ t : (\alpha(P) - H)^* \mid (\forall n : \mathbb{N} \bullet \exists u : H^* \bullet n < \#u \wedge s^\frown u \in Traces(P)) \bullet (s \restriction (\alpha(P) - H))^\frown t\}$: If s is a divergence of P, $s^\frown u \in Traces(P)$ follows for *any* sequence u of events, so s always possesses an unbounded continuation of consecutive H-events.

$$(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}\ \mathbf{sat}\ S'(s, R)$$

$$\begin{aligned}
&S'(s,R) \equiv_{df}\\
&\quad (\exists\, u : \{\!|\ app_tx, app_rc, a_1, b_0, b_1\ |\!\}^*;\\
&\qquad R_{tx} : \mathbb{P}\{\!|\ app_tx\ |\!\};\ RTX_{a_1} : \mathbb{P}\{\!|\ a_1\ |\!\};\\
&\qquad RDCP_{a_1} : \mathbb{P}\{\!|\ a_1\ |\!\};\ RDCP_{b_0} : \mathbb{P}\{\!|\ b_0\ |\!\};\ RDCP_{b_1} : \mathbb{P}\{\!|\ b_1\ |\!\};\\
&\qquad RRC_{b_0} : \mathbb{P}\{\!|\ b_0\ |\!\};\ RRC_{b_1} : \mathbb{P}\{\!|\ b_1\ |\!\};\ R_{rc} : \mathbb{P}\{\!|\ app_rc\ |\!\} \bullet\\
&\qquad\qquad s = u \restriction \{\!|\ app_tx, app_rc\ |\!\}\\
&\qquad\qquad \wedge\\
&\qquad\qquad R = R_{tx} \cup R_{rc}\\
&\qquad\qquad \wedge\\
&\qquad\qquad RTX_{a_1} \cup RDCP_{a_1} = \{\!|\ a_1\ |\!\}\\
&\qquad\qquad \wedge\\
&\qquad\qquad RDCP_{b_0} \cup RRC_{b_0} = \{\!|\ b_0\ |\!\}\\
&\qquad\qquad \wedge\\
&\qquad\qquad RDCP_{b_1} \cup RRC_{b_1} \{\!|\ b_1\ |\!\}\\
&\qquad\qquad \wedge\\
&\qquad\qquad S_{TX}(u \restriction \{\!|\ app_tx, a_1\ |\!\}, R_{tx} \cup RTX_{a_1})\\
&\qquad\qquad \wedge\\
&\qquad\qquad (S_{DCP} \wr \Delta_{DCP})(u \restriction \{\!|\ a_1, b_0, b_1\ |\!\}, RDCP_{a_1} \cup RDCP_{b_0} \cup RDCP_{b_1})\\
&\qquad\qquad \wedge\\
&\qquad\qquad S_{RC}(u \restriction \{\!|\ b_0, b_1, app_rc\ |\!\}, RRC_{b_0} \cup RRC_{b_1} \cup R_{rc}))
\end{aligned}$$

holds.

The validity of proof obligations 1,2 and 3 implies that the premises of Lemma 6 are fulfilled for each process pair

$$(P, Q) \in \{(ABFTX, ABFRC), ((ABFTX \parallel ABFRC), ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}))\}$$

Applying the $\parallel$-law of Lemma 6 to each of these pairs yields the assertion

$$(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)\ \mathbf{sat}\ S''(s, R)$$

$$\begin{aligned}
&S''(s,R) \equiv_{df}\\
&\quad (\exists\, X : \mathbb{P}\{\!|\ app_tx, a_1\ |\!\};\ Y : \mathbb{P}\{\!|\ a_1, b_0, b_1\ |\!\};\ Z : \mathbb{P}\{\!|\ b_0, b_1, app_rc\ |\!\} \bullet\\
&\qquad\qquad R = X \cup Y \cup Z\\
&\qquad\qquad \wedge\\
&\qquad\qquad S_{TX}(s \restriction \{\!|\ app_tx, a_1\ |\!\}, X)\\
&\qquad\qquad \wedge\\
&\qquad\qquad (S_{DCP} \wr \Delta_{DCP})(s \restriction \{\!|\ a_1, b_0, b_1\ |\!\}, Y)\\
&\qquad\qquad \wedge\\
&\qquad\qquad S_{RC}(s \restriction \{\!|\ b_0, b_1, app_rc\ |\!\}, Z))
\end{aligned}$$

From the validity of proof obligation 4 follows that the law [44, p. 125]

$$\frac{P\ \mathbf{sat}\ S(s, R)}{P \setminus H\ \mathbf{sat}\ (\exists\, u : \alpha(P)^* \bullet s = u \restriction (\alpha(P) - H) \wedge S(u, R \cup H))}\quad [\ NODIV(H)\ ]$$

holds for $P = (ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)$ and $H = \{\!|\ a_1, b_0, b_1\ |\!\}$ and $S(u, R \cup H) = S''(u, R \cup \{\!|\ a_1, b_0, b_1\ |\!\})$.

Application of this law with the corresponding substitutions in S'' results in

$(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}$ **sat** $S'''(s, R)$

$$
\begin{array}{l}
S'''(s,R) \equiv_{df} \\
\quad (\exists\, u : \{\!|\ app\_tx, app\_rc, a_1, b_0, b_1\ |\!\}^*; \\
\qquad X : \mathbb{P}\{\!|\ app\_tx, a_1\ |\!\};\ Y : \mathbb{P}\{\!|\ a_1, b_0, b_1\ |\!\};\ Z : \mathbb{P}\{\!|\ b_0, b_1, app\_rc\ |\!\} \bullet \\
\qquad\qquad s = u \upharpoonright \{\!|\ app\_tx, app\_rc\ |\!\} \\
\qquad\qquad \wedge \\
\qquad\qquad R \cup \{\!|\ a_1, b_0, b_1\ |\!\} = X \cup Y \cup Z \\
\qquad\qquad \wedge \\
\qquad\qquad S_{TX}(u \upharpoonright \{\!|\ app\_tx, a_1\ |\!\}, X) \\
\qquad\qquad \wedge \\
\qquad\qquad (S_{DCP} \wr \Delta_{DCP})(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}, Y) \\
\qquad\qquad \wedge \\
\qquad\qquad S_{RC}(u \upharpoonright \{\!|\ b_0, b_1, app\_rc\ |\!\}, Z))
\end{array}
$$

Next we re-write the specification $S'''(s, R)$ using disjoint refusal subsets decomposed according to the channels involved: The set X in $S'''(s, R)$ may be decomposed into $X = R_{tx} \cup RTX_{a_1}$, where R_{tx} is of type $\mathbb{P}\{\!|\ app\_tx\ |\!\}$ and RTX_{a_1} of type $\mathbb{P}\{\!|\ a_1\ |\!\}$. Using analogous decompositions for R, Y, Z results in

$$
\begin{array}{l}
S'''(s,R) \equiv \\
\quad (\exists\, u : \{\!|\ app\_tx, app\_rc, a_1, b_0, b_1\ |\!\}^*; \\
\qquad R_{tx} : \mathbb{P}\{\!|\ app\_tx\ |\!\};\ RTX_{a_1} : \mathbb{P}\{\!|\ a_1\ |\!\}; \\
\qquad RDCP_{a_1} : \mathbb{P}\{\!|\ a_1\ |\!\};\ RDCP_{b_0} : \mathbb{P}\{\!|\ b_0\ |\!\};\ RDCP_{b_1} : \mathbb{P}\{\!|\ b_1\ |\!\}; \\
\qquad RRC_{b_0} : \mathbb{P}\{\!|\ b_0\ |\!\};\ RRC_{b_1} : \mathbb{P}\{\!|\ b_1\ |\!\};\ R_{rc} : \mathbb{P}\{\!|\ app\_rc\ |\!\} \bullet \\
\qquad\qquad s = u \upharpoonright \{\!|\ app\_tx, app\_rc\ |\!\} \\
\qquad\qquad \wedge \\
\qquad\qquad R = R_{tx} \cup R_{rc} \\
\qquad\qquad \wedge \\
\qquad\qquad R_{tx} \cup R_{rc} \cup \{\!|\ a_1, b_0, b_1\ |\!\} = \\
\qquad\qquad\quad R_{tx} \cup RTX_{a_1} \cup RDCP_{a_1} \cup RDCP_{b_0} \cup RDCP_{b_1} \cup RRC_{b_0} \cup RRC_{b_1} \cup R_{rc} \\
\qquad\qquad \wedge \\
\qquad\qquad S_{TX}(u \upharpoonright \{\!|\ app\_tx, a_1\ |\!\}, R_{tx} \cup RTX_{a_1}) \\
\qquad\qquad \wedge \\
\qquad\qquad (S_{DCP} \wr \Delta_{DCP})(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}, RDCP_{a_1} \cup RDCP_{b_0} \cup RDCP_{b_1}) \\
\qquad\qquad \wedge \\
\qquad\qquad S_{RC}(u \upharpoonright \{\!|\ b_0, b_1, app\_rc\ |\!\}, RRC_{b_0} \cup RRC_{b_1} \cup R_{rc}))
\end{array}
$$

Exploiting the type information about the refusal subsets shows that

$$
\begin{array}{l}
R_{tx} \cup R_{rc} \cup \{\!|\ a_1, b_0, b_1\ |\!\} = \\
\qquad R_{tx} \cup RTX_{a_1} \cup RDCP_{a_1} \cup RDCP_{b_0} \cup RDCP_{b_1} \cup RRC_{b_0} \cup RRC_{b_1} \cup R_{rc}
\end{array}
$$

is equivalent to

$$RTX_{a_1} \cup RDCP_{a_1} = \{\!|\ a_1\ |\!\} \wedge RDCP_{b_0} \cup RRC_{b_0} = \{\!|\ b_0\ |\!\} \wedge RDCP_{b_1} \cup RRC_{b_1} = \{\!|\ b_1\ |\!\}$$

Inserting this into the above predicate results in $S'''(s, R) \equiv S'(s, R)$ and proves obligation 6.

Proof of Trace Safety. Under the premises of the theorem

$$(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}\ \textbf{sat}\ SAFETRACE(s)$$

holds.

Applying the consequence rule for each assertion, the premises of the theorem yield

$$\begin{array}{l} ABFTX\ \textbf{sat}\ S'_{TX}(s) \\ S'_{TX}(s) \equiv_{df}\ data^*(s \upharpoonright \{\!|\ a_1\ |\!\}) \leq^M val^*(s \upharpoonright \{\!|\ app\_tx\ |\!\}) \wedge \rho(s \upharpoonright \{\!|\ a_1\ |\!\}) = s \upharpoonright \{\!|\ a_1\ |\!\} \\ \\ ABFRC\ \textbf{sat}\ S'_{RC}(s) \\ S'_{RC}(s) \equiv_{df}\ val^*(s \upharpoonright \{\!|\ app\_rc\ |\!\}) \leq^1 data^*(\rho(s \upharpoonright \{\!|\ b_0, b_1\ |\!\})) \\ \\ ((DCP \setminus L_{DCP}) \wr \Delta_{DCP})\ \textbf{sat}\ (S_{DCP} \wr \Delta_{DCP})'(s) \\ (S_{DCP} \wr \Delta_{DCP})'(s) \equiv_{df} \\ \qquad \rho(s \upharpoonright \{\!|\ a_1\ |\!\}) = s \upharpoonright \{\!|\ a_1\ |\!\} \Rightarrow \\ \qquad\qquad (val^*(\rho(s \upharpoonright \{\!|\ b_0, b_1\ |\!\})) \leq^1 ((f \circ data) \times bit)^*(s \upharpoonright \{\!|\ a_1\ |\!\})) \end{array}$$

Applying the assertion established in proof obligation 6 to specification parts not involving refusals results in

$$(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC)\ \textbf{sat}\ S'(s)$$

with

$$\begin{array}{rl} S'(s) \equiv_{df} & S'_{TX}(s \upharpoonright \alpha(ABFTX)) \wedge \\ & (S_{DCP} \wr \Delta_{DCP})'(s \upharpoonright \alpha((DCP \setminus L_{DCP}) \wr \Delta_{DCP})) \wedge S'_{RC}(s \upharpoonright \alpha(ABFRC)) \\ \equiv & S'_{TX}(s \upharpoonright \{\!|\ app\_tx, a_1\ |\!\}) \wedge \\ & (S_{DCP} \wr \Delta_{DCP})'(s \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}) \wedge S'_{RC}(s \upharpoonright \{\!|\ b_0, b_1, app\_rc\ |\!\}) \end{array}$$

where the free variable s is of type $\{\!|\ app\_tx, a_1, b_0, b_1, app\_rc\ |\!\}^*$.

From $S'_{TX}(s \upharpoonright \alpha(ABFTX))$ the premise $\rho(s \upharpoonright \{\!|\ a_1\ |\!\}) = s \upharpoonright \{\!|\ a_1\ |\!\}$ of specification $(S_{DCP} \wr \Delta_{DCP})'(s \upharpoonright \{\!|\ b_0, b_1, app\_rc\ |\!\})$ follows. Therefore $S'(s)$ implies

$$\begin{array}{ll} & val^*(s \upharpoonright \{\!|\ app\_rc\ |\!\}) \\ & \hfill [\text{validity of } S'_{RC}(s \upharpoonright \{\!|\ b_0, b_1, app\_rc\ |\!\})] \\ \leq^1 & data^*(\rho(s \upharpoonright \{\!|\ b_0, b_1\ |\!\})) \\ & \hfill [\text{validity of implication in } (S_{DCP} \wr \Delta_{DCP})'(s \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\})] \\ \leq^1 & (f \circ data)^*(s \upharpoonright \{\!|\ a_1\ |\!\}) \\ & \hfill [\text{validity of } S'_{TX}(s \upharpoonright \{\!|\ app\_tx, a_1\ |\!\})] \end{array}$$

$$\leq^M f^*(val^*(s \upharpoonright \{\!|\ app\_tx\ |\!\}))$$

Now let w be a trace of $(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}$. From the validity of proof obligation 5 we know that this process does not diverge. Therefore the semantics of the hiding operator [44, p. 131] implies that w must be a restriction of a trace s of $(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC)$, that is, $w = s \upharpoonright \{\!|\ app\_tx, app\_rc\ |\!\}$. The above inequality established for s implies

$$\begin{aligned} val^*(w \upharpoonright \{\!|\ app\_rc\ |\!\}) &= val^*(s \upharpoonright \{\!|\ app\_rc\ |\!\}) \\ &\leq^{(M+2)} f^*(val^*(s \upharpoonright \{\!|\ app\_tx\ |\!\})) = f^*(val^*(w \upharpoonright \{\!|\ app\_tx\ |\!\})) \end{aligned}$$

As a consequence $SAFETRACE(w)$ holds for $N = M+2$ and this proves the safety property required for the traces of $(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}$.

Proof of Deadlock Freedom. Under the premises of the theorem

$$(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\} \ \mathbf{sat}\ NOBLOCK(s, R)$$

holds.

Let (s, R) be a failure of $(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}$. From the validity of proof obligation 6 we know that $S'(s, R)$ holds. We have to show that

$$S'(s, R) \Rightarrow NOBLOCK(s, R)$$

Since it has already been established that the process satisfies $SAFETRACE(s)$, we know that $\#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) \leq \#(s \upharpoonright \{\!|\ app\_tx\ |\!\})$. Therefore the proof can be devided into the cases "=" and "<". The trace u and the refusal subsets $R_{tx}, \ldots$ are used below as in the definition of $S'(s, R)$.

Case 1. Suppose $\#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) = \#(s \upharpoonright \{\!|\ app\_tx\ |\!\})$. Then the inequality

$$val^*(u \upharpoonright \{\!|\ app\_rc\ |\!\}) \leq^2 (f \circ data)^*(u \upharpoonright \{\!|\ a_1\ |\!\}) \leq^M f^*(val^*(u \upharpoonright \{\!|\ app\_tx\ |\!\}))$$

established above in the proof of trace safety implies that $\#(u \upharpoonright \{\!|\ a_1\ |\!\}) = \#(u \upharpoonright \{\!|\ app\_tx\ |\!\})$. Since $S'(s, R)$ implies $S_{TX}(u \upharpoonright \{\!|\ app\_tx, a_1\ |\!\}, R_{tx} \cup RTX_{a_1})$, the definition of S_{TX} implies $R_{tx} = \emptyset$. Since $R = R_{tx} \cup R_{rc}$ and also $R_{rc} \cap \{\!|\ app\_tx\ |\!\} = \emptyset$, it follows that $R \cap \{\!|\ app\_tx\ |\!\} = \emptyset$, and this is what we had to show for the validity of $NOBLOCK(s, R)$ in case 1.

Case 2. Suppose $\#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) < \#(s \upharpoonright \{\!|\ app\_tx\ |\!\})$. To prove consistency with $NOBLOCK(s, R)$ it has to be shown that $\{\!|\ app\_rc\ |\!\} \not\subseteq R$. We will instead assume $\{\!|\ app\_rc\ |\!\} \subseteq R$ and prove that this implies $\#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) = \#(s \upharpoonright \{\!|\ app\_tx\ |\!\})$.

$$\{\!|\ app\_rc\ |\!\} \subseteq R$$

$$\text{[definition of } R_{rc},\ S'(s, R) \text{ holds]}$$

$$\Rightarrow \{\!|\ app\_rc\ |\!\} = R_{rc}$$

$$[S'(s, R) \text{ implies } S_{RC}(u \upharpoonright \{\!|\ b_0, b_1, app\_rc\ |\!\}, RRC_{b_0} \cup RRC_{b_1} \cup R_{rc})]$$

$$\Rightarrow \#(u \upharpoonright \{\!|\ app\_rc\ |\!\}) = \#\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\}) \wedge RRC_{b_0} \cup RRC_{b_1} = \emptyset$$

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad [S'(s,R) \text{ implies } RDCP_{b_i} \cup RRC_{b_i} = \{\!|\ b_i\ |\!\}]\\
&\Rightarrow RDCP_{b_0} \cup RDCP_{b_1} = \{\!|\ b_0, b_1\ |\!\}\\
&\qquad\qquad\qquad\qquad [S'(s,R) \text{ implies validity of specification}]\\
&\qquad\qquad [(S_{DCP} \wr \Delta_{DCP})(u \upharpoonright \{\!|\ a_1, b_0, b_1\ |\!\}, RDCP_{a_1} \cup RDCP_{b_0} \cup RDCP_{b_1})]\\
&\Rightarrow RDCP_{a_1} = \emptyset \wedge \#\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\}) = \#(u \upharpoonright \{\!|\ a_1\ |\!\})\\
&\qquad\qquad\qquad\qquad [S'(s,R) \text{ implies } RTX_{a_1} \cup RDCP_{a_1} = \{\!|\ a_1\ |\!\}]\\
&\Rightarrow RTX_{a_1} = \{\!|\ a_1\ |\!\}\\
&\quad [S'(s,R) \text{ implies validity of specification } S_{TX}(u \upharpoonright \{\!|\ app\_tx, a_1\ |\!\}, R_{tx} \cup RTX_{a_1})]\\
&\Rightarrow \#(u \upharpoonright \{\!|\ a_1\ |\!\}) = \#(u \upharpoonright \{\!|\ app\_tx\ |\!\})\\
&\qquad\qquad\qquad\qquad [\#\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\}) = \#(u \upharpoonright \{\!|\ app\_rc\ |\!\}) \wedge\ ]\\
&\qquad\qquad\qquad\qquad [\#\rho(u \upharpoonright \{\!|\ b_0, b_1\ |\!\}) = \#(u \upharpoonright \{\!|\ a_1\ |\!\}) \wedge\ ]\\
&\qquad\qquad\qquad\qquad [\#(u \upharpoonright \{\!|\ a_1\ |\!\}) = \#(u \upharpoonright \{\!|\ app\_tx\ |\!\}) \quad]\\
&\Rightarrow \#(u \upharpoonright \{\!|\ app\_rc\ |\!\}) = \#(u \upharpoonright \{\!|\ app\_tx\ |\!\})
\end{aligned}
$$

This completes the proof of deadlock freedom.

Together with the conjunction rule, the proofs of trace safety and of deadlock freedom imply that

$$
\begin{aligned}
&(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\} \ \textbf{sat}\\
&\qquad\qquad SAFETRACE(s) \wedge NOBLOCK(s,R)
\end{aligned}
$$

This completes the proof of Theorem 2.
□

3.3.6 Sub-System Design

$ABFTX, ABFRC$ are ready for implementation according to their explicit representation as given in Section 3.3.2. To prove that these explicit representations fulfill the specifications $S_{TX}(s,R), S_{RC}(s,R)$ the technique described in Section 3.4 is applied. We skip the proofs, because this verification technique will be explained by means of the further decomposition of DCP.

3.3.7 Process Design of DCP

This is the last decomposition step of the development procedure for DCP, using the architecture

$$
\begin{aligned}
DCP &= \mathcal{A}_{DCP}(NET_0, NET_1, APP_0, APP_1, DCP\_CONTROL)\\
&= (NET_0 \parallel NET_1 \parallel APP_0 \parallel APP_1 \parallel DCP\_CONTROL)
\end{aligned}
$$

with the explicit process representations and alphabets introduced in Section 3.3.2. The normal behaviour is represented in its explicit form by using $DCP\_CONTROL = NORMAL$, and the acceptable behaviour is reflected by the case $DCP\_CONTROL = ACCEPTABLE$.

3.3.8 Verification of Process Design for DCP

A procedure analogous to the techniques applied for verification on system level would associate behavioural specifications for each process $NET_0, \ldots, APP_1, DCP\_CONTROL$ and prove that with the architecture $\mathcal{A}_{DCP}$ the component specifications imply the behavioural specifications of DCP given above. However, we will now introduce a new technique allowing us to perform a larger portion of the correctness proofs by means of model checking. To this end, two explicit representations DCP_N and $DCPA_N$ of *sequential CSP processes in normal form* (cf. Section 3.4) will be presented. Using the techniques introduced in 3.4, it will be shown that DCP_N satisfies the normal behaviour specification $S_{DCP}(s, R)$ and $DCPA_N$ the acceptable behaviour specification $(S_{DCP} \wr \Delta_{DCP})(s, R)$ of the dual computer system. Then we can use the *explicit* process specifications $NET_0, \ldots, APP_1, DCP\_CONTROL$ to prove by means of model checking that

$$DCP_N \sqsubseteq_{FD} \mathcal{A}_{DCP}(NET_0, NET_1, APP_0, APP_1, NORMAL)$$

and

$$DCPA_N \sqsubseteq_{FD} \mathcal{A}_{DCP}(NET_0, NET_1, APP_0, APP_1, ACCEPTABLE)$$

The first refinement relation implies that the detailed design of DCP satisfies the normal behaviour specification $S_{DCP}(s, R)$, and the second refinement relation satisfies the acceptable behaviour specification $(S_{DCP} \wr \Delta_{DCP})(s, R)$.

Normal Behaviour Verification

The sequential normal form process satisfying $S_{DCP}(s, R)$ is given by

$$\begin{aligned} DCP_N = {} & state := -1; \\ & *((state = -1) \& a_1?(x, b) \rightarrow state := 0;\ SKIP \\ & \quad \Box \\ & \quad (state = 0) \& b_0!(f(x), b) \rightarrow state := -1;\ SKIP) \end{aligned}$$

Theorem 3 (Sub-System Verification Obligation – Normal Behaviour)

$$DCP_N \ \textbf{sat}\ S_{DCP}(s, R)$$

□

We will skip the proof, because the concept will be better illustrated by the acceptable behaviour verification below.

Exceptional Behaviour Verification

The sequential normal form process $DCPA_N$ is defined by

$$
\begin{array}{l}
DCPA_N = state := -1; \\
\quad *((state = -1) \& a_1?(x, b) \rightarrow \\
\qquad \textbf{if } true \rightarrow state := 0; \; \Box \; true \rightarrow state := 1; \;\; \textbf{fi}; \; SKIP \\
\quad [] \\
\quad (state = 0) \& b_0!(f(x), b) \rightarrow \\
\qquad \textbf{if } true \rightarrow state := -1; \; \Box \; true \rightarrow state := 3; \;\; \textbf{fi}; \; SKIP \\
\quad [] \\
\quad (state = 1) \& b_1!(f(x), b') \rightarrow \\
\qquad \textbf{if } true \rightarrow state := -1; \; \Box \; true \rightarrow state := 2; \;\; \textbf{fi}; \; SKIP \\
\quad [] \\
\quad (state = 2) \& b_0!(f(x), b') \rightarrow state := -1; \; SKIP \\
\quad [] \\
\quad (state = 3) \& b_1!(f(x), b') \rightarrow state := -1; \; SKIP)
\end{array}
$$

We will now prove that $DCPA_N$ really satisfies the acceptable behaviour specification $(S_{DCP} \wr \Delta_{DCP})(s, R)$.

Theorem 4 (Sub-System Verification Obligation – Acceptable Behaviour)

$$DCPA_N \; \textbf{sat} \; (S_{DCP} \wr \Delta_{DCP})(s, R)$$

Proof.
The proof will be conducted as follows:

1. Transform $DCPA_N$ into its associated nondeterministic sequential program $\nu(DCPA_N)$ as defined in 3.4.3.
2. Prove that $\nu(DCPA_N)$ **sat** $\iota((S_{DCP} \wr \Delta_{DCP})(s, R))$ where this satisfaction relation for sequential programs is defined in 3.4.4.
3. Apply Theorem 6 to deduce that (2.) implies $DCPA_N$ **sat** $(S_{DCP} \wr \Delta_{DCP})(s, R)$.

According to the definition in 3.4.3, the nondeterministic sequential program associated with

$DCPA_N$ is

$$
\begin{array}{l}
\nu(DCPA_N) = state := -1;\ s_v := \langle\ \rangle;\ R_v := \{\!|\ b_0, b_1\ |\!\};\\
\quad \mathbf{do}\\
\quad (\Box_{(x',b')\in\alpha(a_1)}\ (state = -1) \rightarrow\\
\qquad (x, b) := (x', b');\\
\qquad \mathbf{if}\ true \rightarrow state := 0;\ \Box\ true \rightarrow state := 1;\ \ \mathbf{fi};\\
\qquad s_v := s_v {}^\frown \langle a_1.(x', b')\rangle;\\
\qquad R_v := \{\!|\ a_1, b_0, b_1\ |\!\} -\\
\qquad\quad (\mathbf{if}\ state = 0\ \mathbf{then}\ \{b_0.(f(x), b)\}\ \mathbf{else}\ \{b_1.(f(x), b)\}))\\
\quad \Box\ (state = 0) \rightarrow\\
\qquad (z_{b_0}, b_{b_0}) := (f(x), b);\\
\qquad \mathbf{if}\ true \rightarrow state := -1;\ \Box\ true \rightarrow state := 3;\ \ \mathbf{fi};\\
\qquad s_v := s_v {}^\frown \langle b_0.(z_{b_1}, b_{b_1})\rangle;\\
\qquad R_v := \{\!|\ a_1, b_0, b_1\ |\!\} -\\
\qquad\quad (\mathbf{if}\ state = -1\ \mathbf{then}\ \{\!|\ a_1\ |\!\}\ \mathbf{else}\ \{b_1.(f(x), b)\}))\\
\quad \Box\ (state = 1) \rightarrow\\
\qquad (z_{b_1}, b_{b_1}) := (f(x), b);\\
\qquad \mathbf{if}\ true \rightarrow state := -1;\ \Box\ true \rightarrow state := 2;\ \ \mathbf{fi};\\
\qquad s_v := s_v {}^\frown \langle b_1.(z_{b_1}, b_{b_1})\rangle;\\
\qquad R_v := \{\!|\ a_1, b_0, b_1\ |\!\} -\\
\qquad\quad (\mathbf{if}\ state = -1\ \mathbf{then}\ \{\!|\ a_1\ |\!\}\ \mathbf{else}\ \{b_0.(f(x), b)\}))\\
\quad \Box\ (state = 2) \rightarrow\\
\qquad (z_{b_0}, b_{b_0}) := (f(x), b);\\
\qquad state := -1;\\
\qquad s_v := s_v {}^\frown \langle b_0.(z_{b_1}, b_{b_1})\rangle;\\
\qquad R_v := \{\!|\ b_0, b_1\ |\!\}\\
\quad \Box\ (state = 3) \rightarrow\\
\qquad (z_{b_1}, b_{b_1}) := (f(x), b);\\
\qquad state := -1;\\
\qquad s_v := s_v {}^\frown \langle b_1.(z_{b_1}, b_{b_1})\rangle;\\
\qquad R_v := \{\!|\ b_0, b_1\ |\!\}\\
\quad \mathbf{od}
\end{array}
$$

By the definition given in 3.4.4 and by application of the proof theory for sequential non-deterministic programs, establishing $\nu(DCPA_N)$ **sat** $\iota((S_{DCP} \wr \Delta_{DCP})(s, R))$ is equivalent to proving that

$$I_0 \equiv_{df} (\forall\, U : \mathbb{P}\, R_v \bullet (S_{DCP} \wr \Delta_{DCP})(s_v, R_v))$$

is an invariant of the **do** ... **od**-loop of $\nu(DCPA_N)$. We will instead prove that

$$
\begin{aligned}
I \equiv_{df}\ & \rho(s_v \restriction \{\!|\ a_1\ |\!\}) = s_v \restriction \{\!|\ a_1\ |\!\} \Rightarrow \\
& \quad (val^*(\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\})) \leq^1 ((f \circ data) \times bit)^*(s_v \restriction \{\!|\ a_1\ |\!\}) \\
& \quad \wedge \\
& \quad (\{last(s_v), last(front(s_v))\} \subseteq \{\!|\ b_0, b_1\ |\!\} \Rightarrow \\
& \qquad\qquad last(front(front(s_v))) \in \{\!|\ a_1\ |\!\} \wedge R_v \cap \{\!|\ a_1\ |\!\} = \emptyset) \\
& \quad \wedge \\
& \quad (last(s_v) \in \{\!|\ b_0, b_1\ |\!\} \Rightarrow \{\!|\ a_1, b_0, b_1\ |\!\} \not\subseteq R_v) \\
& \quad \wedge \\
& \quad (\#\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\}) < \#(s_v \restriction \{\!|\ a_1\ |\!\}) \Rightarrow \{\!|\ b_0, b_1\ |\!\} \not\subseteq R_v) \\
& \quad \wedge \\
& \quad state \in \{-1, 0, 1, 2, 3\} \\
& \quad \wedge \\
& \quad (state = -1 \Rightarrow val^*(\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\})) = ((f \circ data) \times bit)^*(s_v \restriction \{\!|\ a_1\ |\!\})) \\
& \quad \wedge \\
& \quad (state \in \{0, 1\} \Rightarrow last(s_v) = a_1.(x, b) \wedge \\
& \qquad val^*(\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\})) = front(((f \circ data) \times bit)^*(s_v \restriction \{\!|\ a_1\ |\!\}))) \\
& \quad \wedge \\
& \quad (state = 2 \Rightarrow last(s_v) = b_1.(f(x), b) \wedge \\
& \qquad val^*(\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\})) = ((f \circ data) \times bit)^*(s_v \restriction \{\!|\ a_1\ |\!\})) \\
& \quad \wedge \\
& \quad (state = 3 \Rightarrow last(s_v) = b_0.(f(x), b) \wedge \\
& \qquad val^*(\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\})) = ((f \circ data) \times bit)^*(s_v \restriction \{\!|\ a_1\ |\!\})))
\end{aligned}
$$

is an invariant, which obviously implies I_0.

Trivially, I holds after the initialisation $state := -1;\ s_v := \langle\ \rangle;\ R_v := \{\!|\ b_0, b_1\ |\!\};$ of $\nu(DCPA_N)$. We will now show for each branch of the **do** ... **od**-loop that, provided I holds on entry, it will also hold on exit of this branch.

Case $state = -1$**.** We fix s_0 as the pre-state of s_v on loop entry and assume the validity of I. Then $I[s_0/s] \wedge state = -1$ holds on loop entry, which implies

$$val^*(\rho(s_0 \restriction \{\!|\ b_0, b_1\ |\!\})) = ((f \circ data) \times bit)^*(s_0 \restriction \{\!|\ a_1\ |\!\})$$

Applying the proof rules for assignment, sequential composition and the **if** ... **fi**-statement results in the post condition

$$
\begin{aligned}
post_{(-1)} \equiv\ & val^*(\rho(s_0 \restriction \{\!|\ b_0, b_1\ |\!\})) = ((f \circ data) \times bit)^*(s_0 \restriction \{\!|\ a_1\ |\!\}) \\
& \wedge \\
& s_v = s_0 {}^\frown \langle a_1.(x, b)\rangle \\
& \wedge \\
& (state = 0 \wedge R_v = \{\!|\ a_1, b_0, b_1\ |\!\} - \{b_0.(f(x), b)\} \\
& \qquad \vee\ state = 1 \wedge R_v = \{\!|\ a_1, b_0, b_1\ |\!\} - \{b_1.(f(x), b)\})
\end{aligned}
$$

on exiting the branch $state = -1$. Now $post_{(-1)}$ implies

$$\begin{aligned} val^*(\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\})) &= val^*(\rho((s_0 \frown \langle a_1.(x,b)\rangle) \restriction \{\!|\ b_0, b_1\ |\!\})) \\ &= val^*(\rho(s_0 \restriction \{\!|\ b_0, b_1\ |\!\})) \\ &= ((f \circ data) \times bit)^*(s_0 \restriction \{\!|\ a_1\ |\!\}) \\ &= front(((f \circ data) \times bit)^*((s_0 \frown \langle a_1.(x,b)\rangle) \restriction \{\!|\ a_1\ |\!\})) \\ &= front(((f \circ data) \times bit)^*(s_v \restriction \{\!|\ a_1\ |\!\})) \end{aligned}$$

Since the after-state of R_v implies $\{\!|\ b_0, b_1\ |\!\} \not\subseteq R_v$, the validity of I on exiting the $(state = -1)$-branch is established.

Case $state = 0$**.** Again, s_0 is fixed as the pre-state of s_v on loop entry and the validity of I is assumed, so that $I[s_0/s] \wedge state = 0$ holds, which implies

$$last(s_0) = a_1.(x,b) \wedge val^*(\rho(s_0 \restriction \{\!|\ b_0, b_1\ |\!\})) = front(((f \circ data) \times bit)^*(s_0 \restriction \{\!|\ a_1\ |\!\}))$$

Applying the proof rules for assignment, sequential composition and the **if** ... **fi**-statement now results in the post condition

$$\begin{aligned} post_0 \equiv\ & last(s_0) = a_1.(x,b) \\ & \wedge \\ & val^*(\rho(s_0 \restriction \{\!|\ b_0, b_1\ |\!\})) = front(((f \circ data) \times bit)^*(s_0 \restriction \{\!|\ a_1\ |\!\})) \\ & \wedge \\ & s_v = s_0 \frown \langle b_0.(f(x),b)\rangle \\ & \wedge \\ & (state = -1 \wedge R_v = \{\!|\ b_0, b_1\ |\!\} \\ & \qquad \vee\ state = 3 \wedge R_v = \{\!|\ a_1, b_0, b_1\ |\!\} - \{b_1.(f(x),b)\}) \end{aligned}$$

on exiting the branch $state = 0$.

We assume that $\rho(s_v \restriction \{\!|\ a_1\ |\!\}) = s_v \restriction \{\!|\ a_1\ |\!\}$ holds, for otherwise there is nothing to prove about I. Then also $\rho(s_0 \restriction \{\!|\ a_1\ |\!\}) = s_0 \restriction \{\!|\ a_1\ |\!\}$ is valid, and therefore the second conjunct in $post_0$ implies

$$\begin{aligned} bit(last(\rho(s_0 \restriction \{\!|\ b_0, b_1\ |\!\}))) &= bit(last(front(s_0 \restriction \{\!|\ a_1\ |\!\}))) \\ &\neq bit(last(s_0 \restriction \{\!|\ a_1\ |\!\})) \\ &= b \end{aligned}$$

Therefore, by definition of ρ, we can calculate

$$\begin{aligned} val^*(\rho(s_v \restriction \{\!|\ b_0, b_1\ |\!\})) &= val^*(\rho((s_0 \frown \langle b_0.(f(x),b)\rangle) \restriction \{\!|\ b_0, b_1\ |\!\})) \\ &= val^*(\rho(s_0 \restriction \{\!|\ b_0, b_1\ |\!\}) \frown \langle b_0.(f(x),b)\rangle) \\ &= front(((f \circ data) \times bit)^*(s_0 \restriction \{\!|\ a_1\ |\!\})) \frown \langle (f(x),b)\rangle \\ &= ((f \circ data) \times bit)^*(s_v \restriction \{\!|\ a_1\ |\!\}) \end{aligned}$$

This establishes I for $state \in \{-1, 3\}$, which are the possible after-states of $state$ in this branch.

Case $state = 1$**.** In analogy to case $state = 0$.

Case $state = 2$**.** In analogy to case $state = 3$.

Case $state = 3$**.** Again, s_0 is fixed as the pre-state of s_v on loop entry and the validity of I is assumed, so that $I[s_0/s] \wedge state = 3$ holds, which implies

$$last(s_0) = b_0.(f(x), b) \wedge val^*(\rho(s_0 \restriction \{\!\mid b_0, b_1 \mid\!\})) = ((f \circ data) \times bit)^*(s_0 \restriction \{\!\mid a_1 \mid\!\})$$

Applying the proof rules for assignment, sequential composition and the **if** ... **fi**-statement now results in the post condition

$$\begin{aligned} post_3 \equiv\ & last(s_0) = b_0.(f(x), b) \\ & \wedge \\ & val^*(\rho(s_0 \restriction \{\!\mid b_0, b_1 \mid\!\})) = ((f \circ data) \times bit)^*(s_0 \restriction \{\!\mid a_1 \mid\!\}) \\ & \wedge \\ & s_v = s_0 \frown \langle b_1.(f(x), b) \rangle \\ & \wedge \\ & (state = -1 \wedge R_v = \{\!\mid b_0, b_1 \mid\!\}) \end{aligned}$$

on exiting the branch $state = 3$.

This time $post_3$ and the definition of ρ implies

$$\begin{aligned} val^*(\rho(s_v \restriction \{\!\mid b_0, b_1 \mid\!\})) &= val^*(\rho((front(s_0) \frown \langle b_0.(f(x), b), b_0.(f(x), b) \rangle) \restriction \{\!\mid b_0, b_1 \mid\!\})) \\ &= val^*(\rho(s_0 \restriction \{\!\mid b_0, b_1 \mid\!\})) \\ &= ((f \circ data) \times bit)^*(s_0 \restriction \{\!\mid a_1 \mid\!\}) \end{aligned}$$

This establishes I for the new value $state = -1$.

This completes the proof of the assertion $\nu(DCPA_N)$ **sat** $\iota((S_{DCP} \wr \Delta_{DCP})(s, R))$. Now Theorem 6 shows that also $DCPA_N$ **sat** $(S_{DCP} \wr \Delta_{DCP})(s, R)$ holds, and this completes the proof of the theorem.
□

The verification process is now completed by

Theorem 5 *The explicit process representations NET_i, APP_i, $i = 0, 1$, NORMAL and ACCEPTABLE defined in Section 3.3.2 satisfy*

$$DCP_N \sqsubseteq_{FD} \mathcal{A}_{DCP}(NET_0, NET_1, APP_0, APP_1, NORMAL) \setminus (\alpha(DCP) - \{\!\mid a_1, b_0, b_1 \mid\!\})$$

and

$$DCPA_N \sqsubseteq_{FD} \mathcal{A}_{DCP}(NET_0, NET_1, APP_0, APP_1, ACCEPTABLE) \setminus (\alpha(DCP) - \{\!\mid a_1, b_0, b_1 \mid\!\})$$

Proof.
By model checking using FDR [27].
□

3.4 Verification of Behavioural Properties for Sequential Processes

Following the specification and verification concept for dependable systems introduced in Chapter 2 it is necessary to switch from implicit specifications to explicit process representations, as soon as the refinement process has reached the stage where implementable units are generated. In this section a technique for the verification of the refinement steps turning implicitly defined black boxes into explicit representations as sequential CSP processes is introduced. Given a specification P **sat** $S(s, R)$ in the failures model and a specific explicit representation (to be introduced in 3.4.2) of P as a communicating sequential process, we can use a proof theory about nondeterministic *sequential* programs to establish that the explicit representation of P really satisfies $S(s, R)$. The method was already introduced and motivated in an intuitive way in [74]. We will now present its formal justification.

3.4.1 Sequential Nondeterministic Programs

We will use the syntax and semantics of *Sequential Nondeterministic Programs* S as introduced by Apt and Olderog [5, pp. 107]. S has the syntax

$$\begin{aligned} S ::= {} & \mathit{skip} \mid u := t \mid S_1 \,;\, S_2 \mid \\ & \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } B \textbf{ do } S_1 \textbf{ od} \mid \\ & \textbf{if } \Box_{i=1}^{n} B_i \rightarrow S_i \textbf{ fi} \mid \ \textbf{do } \Box_{i=1}^{n} B_i \rightarrow S_i \textbf{ od} \end{aligned}$$

The operational semantics of assignment and deterministic sequential control structures is standard [5, p. 60]. $\textbf{if } \Box_{i=1}^{n} B_i \rightarrow S_i \textbf{ fi}$ is the *nondeterministic alternative command*: If the *guard* (i. e., boolean expression) B_i evaluates to *true*, the branch S_i may be executed. If several guards are *true*, the decision is nondeterministic. If all guards evaluate to *false*, the command fails. In our context the *nondeterministic repetitive command* $\textbf{do } \Box_{i=1}^{n} B_i \rightarrow S_i \textbf{ od}$ is of specific importance, since it is directly related to the normal form structure of sequential CSP processes to be introduced in 3.4.2. Its operational semantics is given by the transition axioms

$$< \textbf{do } \Box_{i=1}^{n} B_i \rightarrow S_i \textbf{ od}, \sigma > \longrightarrow < S_i \,;\, \textbf{do } \Box_{i=1}^{n} B_i \rightarrow S_i \textbf{ od}, \sigma >$$

if $\sigma \models B_i$ and $i \in \{1, \ldots, n\}$ and

$$< \textbf{do } \Box_{i=1}^{n} B_i \rightarrow S_i \textbf{ od}, \sigma > \longrightarrow < E, \sigma >$$

if $\sigma \models \bigwedge_{i=1}^{n} \neg\, B_i$: If a guard B_i evaluates to *true* in the actual state σ, the corresponding branch S_i may be executed, and the resulting change of the state σ will be as induced by the operational semantics of S_i. As in the interpretation of the nondeterministic alternative command, several guards evaluating to *true* lead to a nondeterministic decision. The command will be repeated, as long as at least one guard evaluates to *true* at the beginning of the loop. If all guards evaluate to *false* in state σ, the repetitive statement terminates without changing the state (E denotes the empty program). By *SN* we denote the set of all sequential nondeterministic programs.

In addition to the assignment of a formal semantics, Apt and Olderog have developed a sound proof theory for sequential nondeterministic programs allowing to reason about programs by means of proof rules over pre- and postconditions [5, pp. 114]. One of the main advantages of the correspondence between CSP processes and sequential programs to be established in the subsequent sections consists in the possibility to use this proof theory and the tools supporting such "sequential reasoning" in order to verify behavioural properties of CSP processes. However, we will only use the transition axioms of the semantics to define and prove the relationship between CSP processes and sequential programs.

3.4.2 Sequential CSP Processes in Normal Form

Let us now turn to sequential CSP processes P with local variables, as introduced by Hoare [44, pp. 171]. We say that P has a *(second) normal form* in the sense of Apt and Olderog [5, 2, 4], if it can be written as

$$\begin{aligned} P ::= {} & S_0\,; \\ & (\bigvee_{(c,i):\Gamma_{IN}\cup\Gamma_{OUT}} B_{c,i}) * (\ \Box_{(c,i):\Gamma_{IN}}(B_{c,i}\&c?x_{c,i} \rightarrow S_{c,i}\,;\mathit{SKIP}) \\ & \qquad\qquad\qquad\qquad\quad \Box_{(c,i):\Gamma_{OUT}}(B_{c,i}\&c!x_{c,i} \rightarrow S_{c,i}\,;\mathit{SKIP}))\,; \\ & \mathit{SKIP} \end{aligned}$$

In this syntactic expression, S_0 and $S_{c,i}$ are sequential nondeterministic program parts in the syntax and interpretation as defined in 3.4.1. They operate on local variables of P, but do not access any CSP channels. $B * Q$ is the loop construct introduced by Hoare [44, p. 186] for sequential CSP processes: As long as the boolean expression B evaluates to *true*, the sequential CSP process Q will be repeated after each termination. If B is false, the construct terminates, producing the termination event $\surd$. The externally visible actions of P are channel events produced on input channels $c \in \mathit{IN}$ and output channels $c \in \mathit{OUT}$, with $\mathit{IN} \cap \mathit{OUT} = \emptyset$, so that the alphabet of P is $\alpha(P) = \bigcup_{c:\mathit{IN}\cup \mathit{OUT}}\{\!|\ c\ |\!\}$. $B_{c,i}\&c?x_{c,i}$ and $B_{c,i}\&c!x_{c,i}$ denote *guarded communications* with boolean expressions $B_{c,i}$ and defined by the syntactic equivalence

$$B_{c,i}\&c?x_{c,i} \rightarrow S_{c,i}\,;\mathit{SKIP} \equiv_{df} \textbf{if } B_{c,i} \textbf{ then } c?x_{c,i} \rightarrow S_{c,i}\,;\mathit{SKIP} \textbf{ else } \mathit{STOP}$$

The set Γ_{IN} comprises the pairs (c, i) where c is an input channel of P, appearing in a guarded communication $B_{c,i}\&c?x_{c,i}$. Analogously, Γ_{OUT} is defined for the output channels of P. Observe that a channel c may appear more than once in the alternative command at the beginning of the loop, if more than one guard $B_{c,i}$ is associated with c.

With these explanations in mind, the structure of P may be motivated as follows: An execution of P starts with an initialisation phase S_0 assigning values to local variables, including guards. Then P will perform a loop (the *main loop*), where possible communications with other processes may only occur at the beginning of each loop entry. P refuses to communicate on a certain channel c, if all associated guards $B_{c,i}$ evaluate to *false*. If more than one guard is *true* and the corresponding channels are not refused by the environment, the result of the alternative command is nondeterministic. If an output $c!x_{c,i}$ takes place, the communication partner receives the value $\sigma(x_{c,i})$ of the local variable $x_{c,i}$ according to the after-state σ of the previous cycle or the initialisation phase, respectively. An input event associated with expression $in?x_{in,i}$ has the effect of assigning the value passed by the communication partner over channel in to the local variable $x_{in,i}$ of P. After a communication $c.x_{c,i}$ the corresponding sequential program part $S_{c,i}$ is executed locally without further communications.

$S_{c,i}$ may read and change all local variables, including the guards and output variables $x_{c,i}$. If all guards evaluate to *false* after the initialisation or after the last cycle, P will terminate. We consider *SKIP* as a normal form process, equivalent to $SKIP;\ (false) * SKIP;\ SKIP$

While being more restrictive than the general sequential CSP processes covered by the proof system developed by Apt, Francez and de Roever [3], distributed programs consisting of processes in second normal form have the advantage of a simpler proof theory and of a more obvious correspondence to nondeterministic sequential programs (see [5, 2, 4]). Normal form processes are also of specific interest from a practical point of view, because the technique of isolating external communications at the beginning of a main loop is a widely adopted style for parallel programming.

For the remaining part of this chapter, we will consider CSP processes P over a fixed alphabet $A = \alpha(P)$.

Our next objective is to relate a state $\tau \in \Sigma(P)$ of local variables of P to the communication behaviour. To this end, let $\Sigma(P, s)$ denote the collection of all *variable states* possible in *process* state P/s, when evaluated before entering the main loop or, if $last(s) = \surd$, on termination. Applying the semantic rules for sequential CSP processes in combination with the semantics of the nondeterministic sequential program parts $S_0, S_{c,i}$, the set can be inductively defined by[5]

$$\Sigma(P, \langle\,\rangle) = \{\tau, \tau' : \Sigma(P) \mid < S_0, \tau > \longrightarrow^* < E, \tau' > \bullet\ \tau'\}$$

$$\begin{aligned}\Sigma(P, s^\frown\langle c.v\rangle) = &\ \mathbf{if}\ c \in IN \\ &\ \mathbf{then}\ \{\tau : \Sigma(P, s);\ \tau' : \Sigma(P) \mid \\ &\qquad (\exists\, i \bullet \tau \models B_{c,i}) \wedge < x_{c,i} := v;\ S_{c,i}, \tau > \longrightarrow^* < E, \tau' > \bullet\ \tau'\} \\ &\ \mathbf{else}\ \ \{\tau : \Sigma(P, s);\ \tau' : \Sigma(P) \mid \\ &\qquad (\exists\, i \bullet \tau \models B_{c,i} \wedge \tau(x_{c,i}) = v) \wedge < S_{c,i}, \tau > \longrightarrow^* < E, \tau' > \bullet\ \tau'\}\end{aligned}$$

$$\Sigma(P, s^\frown\langle \surd \rangle) = \{\tau : \Sigma(P, s) \mid \tau \models (\wedge_{(c,i) \in \Gamma_{IN} \cup \Gamma_{OUT}} \neg\, B_{c,i}) \bullet \tau\}$$

In this definition, $\longrightarrow^*$ is the reflexive and transitive closure of the state transition relation $\longrightarrow$: $< S_0, \sigma > \longrightarrow^* < S', \sigma' >$ indicates the existence of a sequence of state transitions, starting with S_0 in the state σ and reaching S' in the new state σ'. $\Sigma(P, \langle\,\rangle)$ is the set of possible after-states of S_0 (recall that E denotes the empty program). In the definition of $\Sigma(P, s^\frown\langle c.v\rangle)$ observe that the communication $c.v$ can only take place if the corresponding guard $B_{c,i}$ evaluates to *true* before entering the main loop. This is expressed by the requirement $\tau \models B_{c,i}$, where τ is a pre-state on main loop entry. The effect of the input $c.v$ is the assignment of value v to the corresponding channel variable $x_{c,i}$. Therefore the elements of $\Sigma(P, s^\frown\langle c.v\rangle)$ are the possible after-states of the program $x_{c,i} := v;\ S_{c,i}$. If $c.v$ is an output event, this presupposes again the existence of a corresponding guard evaluating to *true* in the pre-state τ. Additionally, the output variable $x_{c,i}$ must have value v is this state. Since output does not change the local variable state, we now collect the after-states of $< S_{c,i}, \tau >$ in $\Sigma(P, s^\frown\langle c.v\rangle)$. P terminates if and only if all guards evaluate to *false* in the actual state

[5]Recall that we use the Z notation for set comprehension: $\{x : T \mid cond(x) \bullet expr(x)\}$ is the set of all elements defined by the expressions $expr(x)$, where x ranges over all elements of type T, satisfying condition $cond(x)$.

τ on main loop entry. In this case no further assignments take place, so that such a τ is also a valid final state.

Making use of the definition of $\Sigma(P, s)$, the following lemma describes the relationship between traces, refusals and local variable states:

Lemma 8 *Let $s \in \mathit{Traces}(P)$ and $\mathit{last}(s) \neq \surd$. Then*

1. *The set $[P/s]^0$ of events possible in process state P/s is given by*

$$
\begin{aligned}
[P/s]^0 = \{\tau : \Sigma(P,s);\ (in, i) : \Gamma_{IN};\ y : \alpha(in) \mid \tau \models B_{in,i} \bullet in.y\} \cup \\
\{\tau : \Sigma(P,s);\ (out, i) : \Gamma_{OUT} \mid \tau \models B_{out,i} \bullet out.\tau(x_{out,i})\} \cup \\
\{\tau : \Sigma(P,s) \mid \tau \models (\textstyle\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i}) \bullet \surd\}
\end{aligned}
$$

2. *The refusals of P/s are given by*

$$
\begin{aligned}
\mathit{Ref}(P/s) = \textstyle\bigcup_{\tau:\Sigma(P,s)} (&\mathbf{if}\ \tau \models (\textstyle\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i}) \\
&\mathbf{then}\ \mathbb{P}(A - \{\surd\}) \\
&\mathbf{else}\ \ \mathbb{P}(A - (\textstyle\bigcup_{(in,j):\{(c,i):\Gamma_{IN} \mid \tau\models B_{in,i}\}} \{\!|\ in\ |\!\} \cup \\
&\qquad\qquad \textstyle\bigcup_{(out,j):\{(c,i):\Gamma_{OUT} \mid \tau\models B_{out,i}\}} \{out.\tau(x_{out,j})\})))
\end{aligned}
$$

Proof.
Part (1.) follows directly from the semantics of the main loop construct and the semantics of guarded communications.

For part 2, observe that the behaviour of P/s is nondeterministically defined by its internal states $\tau \in \Sigma(P, s)$, because each τ determines the evaluation of the communication guards. Since the internal variable state cannot be influenced by the environment, P/s can be written as an internal choice ranging over the possible states τ:

$$
\begin{aligned}
P/s = \sqcap_{\tau:\Sigma(P,s)} (&\mathit{init}(\tau); \\
&\mathbf{if}\ (\textstyle\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i}) \\
&\mathbf{then}\ \mathit{SKIP} \\
&\mathbf{else}\ ((\ \square_{(c,i):\Gamma_{IN}} (B_{c,i} \& c?x_{c,i} \to S_{c,i};\ \mathit{SKIP}) \\
&\qquad\quad \square_{(c,i):\Gamma_{OUT}} (B_{c,i} \& c!x_{c,i} \to S_{c,i};\ \mathit{SKIP})); \\
&\qquad (\textstyle\bigvee_{(c,i):\Gamma_{IN}\cup\Gamma_{OUT}} B_{c,i})* \\
&\qquad\ (\ \square_{(c,i):\Gamma_{IN}} (B_{c,i} \& c?x_{c,i} \to S_{c,i};\ \mathit{SKIP}) \\
&\qquad\quad \square_{(c,i):\Gamma_{OUT}} (B_{c,i} \& c!x_{c,i} \to S_{c,i};\ \mathit{SKIP})); \\
&\qquad \mathit{SKIP}))
\end{aligned}
$$

In this expression, $\mathit{init}(\tau')$ denotes the sequence of assignments $x := \tau'(x)$ for each variable x of P. If all the guards evaluate to *false*, P/s will terminate. Otherwise, the main loop will be executed at least once, as shown in the **else**-branch.

Observe that STOP is a unit of $\square$, so that branches with guards evaluating to *false* do not contribute to the alternative construct. Therefore we calculate

$$
\begin{aligned}
&\mathit{Ref}(P) = \\
&\qquad\qquad\qquad\qquad [\mathit{Ref}(\mathit{SKIP}) = \mathbb{P}(A - \{\surd\}), \text{ semantic rules for } \sqcap \text{ and } \Box] \\
&\quad \bigcup_{\tau:\{\tau':\Sigma(P,s) \mid \tau' \models (\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i})\}} \mathbb{P}(A - \{\surd\}) \\
&\quad \cup \\
&\quad \bigcup_{\tau:\{\tau':\Sigma(P,s) \mid \tau' \models (\bigvee_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} B_{c,i})\}} \\
&\qquad (\bigcap_{(in,j):\{(c,i):\Gamma_{IN} \mid \tau \models B_{c,i}\}} \mathit{Ref}(in?x_{in,j} \to S_{in,j};\ \mathit{SKIP}) \cap \\
&\qquad\qquad \bigcap_{(out,j):\{(c,i):\Gamma_{OUT} \mid \tau \models B_{c,i}\}} \mathit{Ref}(out!\tau(x_{out,j}) \to S_{out,j};\ \mathit{SKIP})) = \\
&\qquad\qquad\qquad\qquad [\text{semantic rules for choice and prefixing}] \\
&\quad \bigcup_{\tau:\{\tau':\Sigma(P,s) \mid \tau' \models (\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i})\}} \mathbb{P}(A - \{\surd\}) \\
&\quad \cup \\
&\quad \bigcup_{\tau:\{\tau':\Sigma(P,s) \mid \tau' \models (\bigvee_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} B_{c,i})\}} \\
&\qquad (\bigcap_{(in,j):\{(c,i):\Gamma_{IN} \mid \tau \models B_{c,i}\}} \mathbb{P}(A - \{\!|\ in\ |\!\}) \cap \\
&\qquad\qquad \bigcap_{(out,j):\{(c,i):\Gamma_{OUT} \mid \tau \models B_{c,i}\}} \mathbb{P}(A - \{out.\tau(x_{out,j})\}) = \\
&\qquad\qquad\qquad\qquad [\text{laws on } \mathbb{P}, \cup, \cap] \\
&\quad \bigcup_{\tau:\{\tau':\Sigma(P,s) \mid \tau' \models (\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i})\}} \mathbb{P}(A - \{\surd\}) \\
&\quad \cup \\
&\quad \bigcup_{\tau:\{\tau':\Sigma(P,s) \mid \tau' \models (\bigvee_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} B_{c,i})\}} \\
&\qquad (\mathbb{P}(A - (\bigcup_{(in,j):\{(c,i):\Gamma_{IN} \mid \tau \models B_{c,i}\}} \{\!|\ in\ |\!\} \cup \\
&\qquad\qquad \bigcup_{(out,j):\{(c,i):\Gamma_{OUT} \mid \tau \models B_{c,i}\}} \{out.\tau(x_{out,j})\})))
\end{aligned}
$$

The last line is equal to the expression used in part (2.) above. This completes the proof of the lemma.
□

3.4.3 Mapping Normal Form Processes to Sequential Programs

Let $\mathit{CSP}_{\mathit{SNF}}$ denote the set of all sequential CSP processes represented in the second normal form introduced above. Let SN be the set of all nondeterministic programs introduced in 3.4.1. We will now define a mapping

$$\nu : \mathit{CSP}_{\mathit{SNF}} \to \mathit{SN}$$

associating with $P \in \mathit{CSP}_{\mathit{SNF}}$ a nondeterministic program $\nu(P)$ which allows to reason about the semantic properties of P by means of the semantics and proof theory valid in SN.

Given

$$
\begin{aligned}
P ::= {} & S_0; \\
& (\bigvee_{(c,i):\Gamma_{IN}\cup\Gamma_{OUT}} B_{c,i}) * (\ \Box_{(c,i):\Gamma_{IN}} (B_{c,i} \& c?x_{c,i} \to S_{c,i};\ \mathit{SKIP}) \\
& \qquad\qquad\qquad\qquad \Box_{(c,i):\Gamma_{OUT}} (B_{c,i} \& c!x_{c,i} \to S_{c,i};\ \mathit{SKIP})); \\
& \mathit{SKIP}
\end{aligned}
$$

we set

$$\begin{array}{ll}\nu(P) ::= & S_0;\ s_v := \langle\ \rangle;\ R_v := r(\Gamma_{IN}, \Gamma_{OUT});\\ & \quad \mathbf{do}\\ & \quad\quad \Box_{(c,i):\Gamma_{IN}, y_c \in \alpha(c)}(B_{c,i} \rightarrow x_{c,i} := y_c;\ S_{c,i};\ s_v := s_v{}^\frown\langle c.y_c\rangle;\ R_v := r(\Gamma_{IN}, \Gamma_{OUT}))\\ & \quad\quad \Box_{(c,i):\Gamma_{OUT}}(B_{c,i} \rightarrow z_{c,i} := x_{c,i};\ S_{c,i};\ s_v := s_v{}^\frown\langle c.z_{c,i}\rangle;\ R_v := r(\Gamma_{IN}, \Gamma_{OUT}))\\ & \quad \mathbf{od}\\ & s_v := s_v{}^\frown\langle \surd\rangle;\ R_v := A.\end{array}$$

where

$$s_v : A^*;\ R_v : \mathbb{P}\, A;\ z_{c,i} : \alpha(c)$$

are fresh local variables of $\nu(P)$, nowhere accessed in the sequential program parts $S_0, S_{c,i}$, and y_c are constants ranging over the channel alphabet $\alpha(c)$. In the assignment of R_v we used the abbreviation

$$\begin{array}{ll} r(\Gamma_{IN}, \Gamma_{OUT}) ::= & \mathbf{if}\ (\bigwedge_{(c,i):\Gamma_{IN}\cup\Gamma_{OUT}} \neg\, B_{c,i})\\ & \mathbf{then}\ (A - \{\surd\})\\ & \mathbf{else}\ \ (A - (\bigcup_{(in,j):\{(c,i):\Gamma_{IN} \mid B_{in,i}\}} \{\!|\ in\ |\!\} \cup \\ & \qquad\qquad \bigcup_{(out,j):\{(c,i):\Gamma_{OUT} \mid B_{out,i}\}} \{out.x_{out,j}\})) \end{array}$$

In correspondence with the P, the **do** … **od** construct is called the *main loop* of $\nu(P)$. Note that consistent with the general definition of ν, the trivial normal form process *SKIP* is mapped to

$$\begin{array}{ll}\nu(SKIP) = & s_v := \langle\ \rangle;\ R_v := A - \{\surd\};\\ & \quad \mathbf{do}\ \mathit{false} \rightarrow \mathit{skip}\ \mathbf{od};\\ & s_v := s_v{}^\frown\langle \surd\rangle;\ R_v := A.\end{array}$$

Before formalising and verifying the relationship between P and $\nu(P)$, let us indicate the intuition behind the construction of $\nu(P)$: The program starts an execution with the same initialisation S_0 as P, additionally assigning initial values to (s_v, R_v). Obviously, the variable s_v corresponds to the trace $s = s_v$ of P: If P/s can perform the communication $c.x_{c,i}$, the corresponding guard $B_{c,i}$ will be *true*. In $\nu(P)$ $c.x_{c,i}$ is replaced by an assignment, and the communication is “recorded” in variable s_v. At each beginning of the **do** … **od**-loop, variable R_v coincides with a maximal refusal of P in state P/s: If all guards evaluate to *false*, expression $r(\Gamma_{IN}, \Gamma_{OUT})$ denotes the maximal refusal of *SKIP*, which is the set of all events except $\surd$. Otherwise $r(\Gamma_{IN}, \Gamma_{OUT})$ denotes the maximal subset of A which does not contain any event where the corresponding guard is *true*.

As a first step towards the precise description of these relationships, we will relate the sets $\Sigma(P, s)$ of P-variable states to corresponding $\nu(P)$-variable states. Let $\Sigma(\nu(P))$ denote the set of all variable states σ of the sequential nondeterministic program $\nu(P)$. Since the sequential program fragments $S_0, S_{c,i}$ in P and $\nu(P)$ are syntactically equal and both are interpreted in the same semantics, the only difference between their state functions consists in the additional variables $s_v, R_v, z_{c,i}$ occurring in $\nu(P)$. Therefore a variable state $\sigma \in \Sigma(\nu(P))$ can be expressed as

$$\sigma = \tau \cup \{s_v \mapsto u, R_v \mapsto U, y_c \mapsto \ldots, z_{c,i} \mapsto \ldots\}$$

with a variable state τ of $\Sigma(P)$. Given $\sigma \in \Sigma(\nu(P))$, we use $\pi_P(\sigma) \in \Sigma(P)$ to denote the domain restriction of σ to the variables occurring in P. For a sequence of events $s \in A^*$ define

$$\Sigma(\nu(P), s) = \{\sigma, \sigma' : \Sigma(\nu(P));\ S' : \{E, S_L\} \mid \\ < \nu(P), \sigma > \longrightarrow^* < S', \sigma' > \wedge\, \sigma'(s_v) = s \bullet \pi_P(\sigma')\}$$

In this expression S_L is an abbreviation for the program fragment

do ... **od**; $s_v := s_v{}^\frown\langle \surd \rangle;\ R_v := A.$

starting with the main loop of $\nu(P)$. $\Sigma(\nu(P), s)$ is the collection of all variable states σ' "recorded" during an execution of $\nu(P)$ either on main loop entry or on termination, such that the variable s_v has the value s in this state. Note that the termination event $\surd$ is only appended to s_v on termination of $\nu(P)$. Therefore

$$last(s) \neq \surd \Rightarrow \\ \Sigma(\nu(P), s) = \{\sigma, \sigma' : \Sigma(\nu(P)) \mid \\ < \nu(P), \sigma > \longrightarrow^* < S_L, \sigma' > \wedge\, \sigma'(s_v) = s \bullet \pi_P(\sigma')\}$$

$$last(s) = \surd \Rightarrow \\ \Sigma(\nu(P), s) = \{\sigma, \sigma' : \Sigma(\nu(P)) \mid \\ < \nu(P), \sigma > \longrightarrow^* < E, \sigma' > \wedge\, \sigma'(s_v) = s \bullet \pi_P(\sigma')\}$$

The following lemma shows that the collection of sets $\Sigma(\nu(P), s)$ is prefix-closed with respect to s in the following sense:

Lemma 9 *If σ' is a state contained in $\Sigma(\nu(P), s)$ and $s'' \leq s$, then there exists $\sigma \in \Sigma(\nu(P))$; $\sigma'' \in \Sigma(\nu(P), s'')$; $S', S'' \in \{E, S_L\}$ such that*

$$< \nu(P), \sigma > \longrightarrow^* < S'', \sigma'' > \longrightarrow^* < S', \sigma' > \\ \wedge \\ \sigma''(s_v) = s''$$

Proof.
Each assignment to s_v in $\nu(P)$ appends a single entry to the pre-state of s_v. Therefore, using induction over s_v and the semantic rules for sequential nondeterministic programs, it is easily established that

$$I \equiv_{df} \#\sigma'(s_v) > 0 \Rightarrow \\ (\exists\, \sigma \in \Sigma(\nu(P));\ \sigma'' \in \Sigma(\nu(P), s'');\ S', S'' \in \{E, S_L\} \bullet \\ < \nu(P), \sigma > \longrightarrow^* < S'', \sigma'' > \longrightarrow^* < S', \sigma' > \\ \wedge \\ \sigma''(s_v) = front(s))$$

holds on each main loop entry and on termination of $\nu(P)$. Repetitive application of I yields the assertion of the lemma.
□

The following lemma establishes the relationship between the sets $\Sigma(P, s)$ and $\Sigma(\nu(P), s)$.

Lemma 10 *If P is a sequential CSP process in normal form and $\nu(P)$ the associated sequential nondeterministic program, then*

1. *If $s \in Traces(P)$, then $\Sigma(P,s) = \Sigma(\nu(P),s)$.*
2. *If $s \in A^* - Traces(P)$, then $\Sigma(\nu(P),s) = \emptyset$.*

Proof.
Proof of Part 1. We use induction over the length of traces s.

Proof Obligation 1.1. Assertion (1.) of the lemma holds for $s = \langle\ \rangle$.

We calculate

$$
\begin{aligned}
&\Sigma(P, \langle\ \rangle) = \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{[definition of } P \text{ and } \Sigma(P,s)] \\
&\quad \{\tau, \tau' : \Sigma(P) \mid < S_0, \tau > \longrightarrow^* < E, \tau' > \\
&\qquad\qquad \bullet\ \tau'\} = \\
&\qquad\qquad\qquad\qquad\qquad\qquad\text{[interpretation of } S_0 \text{ is the same in } P \text{ and } \nu(P)] \\
&\quad \{\sigma, \sigma'' : \Sigma(\nu(P)) \mid < \nu(P), \sigma > \longrightarrow^* < s_v := \langle\ \rangle;\ R_v := r(\Gamma_{IN}, \Gamma_{OUT});\ S_L, \sigma'' > \\
&\qquad\qquad \bullet\ \pi_P(\sigma'')\} = \\
&\qquad\qquad\qquad\qquad\qquad [s_v, R_v \text{ are not contained in the domain of states } \tau \in \Sigma(P)] \\
&\quad \{\sigma, \sigma'' : \Sigma(\nu(P)) \mid < \nu(P), \sigma > \longrightarrow^* < s_v := \langle\ \rangle;\ R_v := r(\Gamma_{IN}, \Gamma_{OUT});\ S_L, \sigma'' > \\
&\qquad\qquad \bullet\ \pi_P(\sigma'' \oplus \{s_v \mapsto \langle\ \rangle, R_v \mapsto \sigma''(r(\Gamma_{IN}, \Gamma_{OUT}))\})\} = \\
&\qquad\qquad\qquad\qquad\qquad\text{[semantic rules for assignment and sequential composition]} \\
&\quad \{\sigma, \sigma' : \Sigma(\nu(P)) \mid < \nu(P), \sigma > \longrightarrow^* < S_L, \sigma' > \wedge\ \sigma'(s_v) = \langle\ \rangle \\
&\qquad\qquad \bullet\ \pi_P(\sigma')\} = \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{[definition of } \nu(P) \text{ and } \Sigma(\nu(P),s)] \\
&\quad \Sigma(\nu(P), \langle\ \rangle)
\end{aligned}
$$

This shows the validity of proof obligation 1.1.

Proof Obligation 1.2. Assume that assertion (1.) of the lemma holds for all traces s with $0 \leq \#s \leq n$. Then the assertion also holds for all $s \frown \langle e \rangle$ satisfying $e \in [P/s]^0$.

Case 1.2.1: Assume $s \frown \langle e \rangle \in Traces(P)$ with $e = \surd$.

Then

$$
\begin{aligned}
&\Sigma(P, s \frown \langle \surd \rangle) = \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{[inductive definition of } \Sigma(P,s)] \\
&\quad \{\tau : \Sigma(P,s) \mid \tau \models (\wedge_{(c,i) \in \Gamma_{IN} \cup \Gamma_{OUT}} \neg B_{c,i}) \bullet \tau\} = \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{[induction hypothesis]} \\
&\quad \{\sigma, \sigma'' : \Sigma(\nu(P)) \mid \\
&\qquad < \nu(P), \sigma > \longrightarrow^* < S_L, \sigma'' > \wedge\ \sigma''(s_v) = s \wedge \sigma'' \models (\wedge_{(c,i) \in \Gamma_{IN} \cup \Gamma_{OUT}} \neg B_{c,i}) \\
&\qquad\qquad \bullet\ \pi_P(\sigma'')\} =
\end{aligned}
$$

[semantics of nondeterministic repetitive command]

$$\{\sigma, \sigma'' : \Sigma(\nu(P)) \mid$$
$$< \nu(P), \sigma > \longrightarrow^* < s_v := s_v \frown \langle \surd \rangle;\ R_v := r(\Gamma_{IN}, \Gamma_{OUT}).,\sigma'' > \wedge\ \sigma''(s_v) = s$$
$$\bullet\ \pi_P(\sigma'')\} =$$

[s_v, R_v are not contained in the domain of states $\tau \in \Sigma(P)$]

$$\{\sigma, \sigma'' : \Sigma(\nu(P)) \mid$$
$$< \nu(P), \sigma > \longrightarrow^* < s_v := s_v \frown \langle \surd \rangle;\ R_v := r(\Gamma_{IN}, \Gamma_{OUT}).,\sigma'' > \wedge\ \sigma''(s_v) = s$$
$$\bullet\ \pi_P(\sigma'' \oplus \{s_v \mapsto s \frown \langle \surd \rangle, R_v \mapsto \sigma''(r(\Gamma_{IN}, \Gamma_{OUT}))\})\} =$$

[Lemma 9, semantic rules for assignment and sequential composition]

$$\{\sigma, \sigma' : \Sigma(\nu(P)) \mid < \nu(P), \sigma > \longrightarrow^* < E, \sigma' > \wedge\ \sigma'(s_v) = s \frown \langle \surd \rangle$$
$$\bullet\ \pi_P(\sigma')\} =$$

[definition of $\Sigma(\nu(P), s)$]

$$\Sigma(\nu(P), s \frown \langle \surd \rangle)$$

This shows the validity of proof obligation 1.2 in case 1.2.1.

Case 1.2.2. Assume $s \frown \langle e \rangle \in \mathit{Traces}(P)$ with $e = \mathit{in}.y_{in}$ and $\mathit{in} \in \mathit{IN}, y_{in} \in \alpha(\mathit{in})$.

Then

$$\Sigma(P, s \frown \langle \mathit{in}.y_{in} \rangle) =$$

[inductive definition of $\Sigma(P, s)$]

$$\{\tau : \Sigma(P, s);\ \tau' : \Sigma(P) \mid$$
$$(\exists\, i \bullet \tau \models B_{in,i}) \wedge < x_{in,i} := y_{in};\ S_{in,i}, \tau > \longrightarrow^* < E, \tau' > \bullet\ \tau'\} =$$

[induction hypothesis]

$$\{\tau'' : \Sigma(\nu(P), s);\ \tau' : \Sigma(P) \mid$$
$$(\exists\, i \bullet \tau'' \models B_{in,i}) \wedge < x_{in,i} := y_{in};\ S_{in,i}, \tau'' > \longrightarrow^* < E, \tau' > \bullet\ \tau'\} =$$

[definition of $\Sigma(\nu(P), s)$, interpretation of]
[$x_{in,i} := y_{in};\ S_{in,i}$ is the same in P and $\nu(P)$]

$$\{\sigma, \sigma', \sigma'' : \Sigma(\nu(P)) \mid$$
$$< \nu(P), \sigma > \longrightarrow^* < S_L, \sigma'' > \wedge\ \sigma''(s_v) = s\ \wedge$$
$$(\exists\, i \bullet \sigma'' \models B_{in,i}) \wedge < x_{in,i} := y_{in};\ S_{in,i}, \sigma'' > \longrightarrow^* < E, \sigma' > \bullet\ \pi_P(\sigma')\} =$$

[semantics of nondeterministic repetitive command]

$$\{\sigma, \sigma', \sigma'' : \Sigma(\nu(P)) \mid$$
$$< \nu(P), \sigma >$$
$$\longrightarrow^* < x_{in,i} := y_{in};\ S_{in,i};\ s_v := s_v \frown \langle \mathit{in}.y_{in} \rangle;\ R_v := r(\Gamma_{IN}.\Gamma_{OUT});\ S_L, \sigma'' > \wedge$$
$$\sigma''(s_v) = s\ \wedge < x_{in,i} := y_{in};\ S_{in,i}, \sigma'' > \longrightarrow^* < E, \sigma' > \bullet\ \pi_P(\sigma')\} =$$

[Lemma 9, semantics of assignment,]
[s_v, R_v are not contained in dom $\pi_P(\sigma')$]

$$
\begin{aligned}
&\{\sigma, \sigma', \sigma'' : \Sigma(\nu(P)) \mid \\
&\quad < \nu(P), \sigma > \\
&\qquad \longrightarrow^{*} < x_{in,i} := y_{in};\ S_{in,i};\ s_v := s_v{}^\frown\langle in.y_{in}\rangle;\ R_v := r(\Gamma_{IN}.\Gamma_{OUT});\ S_L, \sigma'' > \wedge \\
&\quad \sigma''(s_v) = s \wedge \\
&\quad < x_{in,i} := y_{in};\ S_{in,i};\ s_v := s_v{}^\frown\langle in.y_{in}\rangle;\ R_v := r(\Gamma_{IN}.\Gamma_{OUT}), \sigma'' > \\
&\qquad\qquad \longrightarrow^{*} < E, \sigma' \oplus \{s_v \mapsto s^\frown\langle in.y_c\rangle, R_v \mapsto \sigma''(r(\Gamma_{IN}.\Gamma_{OUT}))\} > \\
&\quad \bullet\ \pi_P(\sigma' \oplus \{s_v \mapsto s^\frown\langle in.y_c\rangle, R_v \mapsto \sigma''(r(\Gamma_{IN}.\Gamma_{OUT}))\})\} =
\end{aligned}
$$

[semantics of sequential composition]

$$
\begin{aligned}
&\{\sigma, \sigma', \sigma'' : \Sigma(\nu(P)) \mid \\
&\quad < \nu(P), \sigma > \longrightarrow^{*} < S_L, \sigma' \oplus \{s_v \mapsto s^\frown\langle in.y_c\rangle, R_v \mapsto \sigma''(r(\Gamma_{IN}.\Gamma_{OUT}))\} > \wedge \\
&\quad \sigma''(s_v) = s \\
&\quad \bullet\ \pi_P(\sigma' \oplus \{s_v \mapsto s^\frown\langle in.y_c\rangle, R_v \mapsto \sigma''(r(\Gamma_{IN}.\Gamma_{OUT}))\})\} =
\end{aligned}
$$

[Lemma 9]

$$
\begin{aligned}
&\{\sigma, \sigma' : \Sigma(\nu(P)) \mid \\
&\quad < \nu(P), \sigma > \longrightarrow^{*} < S_L, \sigma' > \wedge\ \sigma'(s_v) = s^\frown\langle in.y_{in}\rangle \\
&\quad \bullet\ \pi_P(\sigma')\} =
\end{aligned}
$$

[definition of $\Sigma(\nu(P), s)$]

$$\Sigma(\nu(P), s^\frown\langle in.y_{in}\rangle)$$

This shows the validity of proof obligation 1.2 in case 1.2.2.

Case 1.2.3. Assume $s^\frown\langle e\rangle \in Traces(P)$ with $e = out.v$ and $out \in OUT$: in analogy to case 1.2.2.

This shows the validity of proof obligation 1.2 and completes the proof of part (1.) of the lemma.

Proof of Part 2. Suppose

$$\sigma' \in \Sigma(\nu(P), s''{}^\frown\langle in.y_{in}\rangle) \wedge s'' \in Traces(P) \wedge s''{}^\frown\langle in.y_{in}\rangle \notin Traces(P)$$

Application of Lemma 9 implies the existence of $\sigma, \sigma'' \in \Sigma(\nu(P))$ such that

$$< \nu(P), \sigma > \longrightarrow^{*} < S_L, \sigma'' > \longrightarrow^{*} < S', \sigma' > \wedge\ \sigma''(s_v) = s''$$

Application of the semantic rules for the nondeterministic repetitive command, assignment and sequential composition implies the existence of a guard $B_{in,i}$ such that $\sigma'' \models B_{in,i}$. Application of part (1.) of this lemma yields the existence of a state $\tau'' = \pi_P(\sigma'') \in \Sigma(P, s'')$ so that $\tau'' \models B_{in,i}$. Application of Lemma 8, (1.) yields the contradiction $in.y_{in} \in [P/s'']^0$. Therefore $\Sigma(\nu(P), s''{}^\frown\langle in.y_{in}\rangle) = \emptyset$ follows. The cases $\sigma' \in \Sigma(\nu(P), s''{}^\frown\langle out.x_{out,i}\rangle)$ and $\sigma' \in \Sigma(\nu(P), s''{}^\frown\langle \surd\rangle)$ are treated analogously. This completes the proof of part (2.).
□

3.4.4 Mapping Behavioural Specifications to Invariants

In addition to the explicit process representation, it is also possible to map behavioural CSP specifications $S(s,R)$ interpreted in the untimed failures model $\mathcal{M}_{UF}$ to predicates interpreted over the variables s_v, R_v for programs in ran $\nu \subseteq SN$: Let $SPEC_{UF}$ be the collection of specifications $S(s,R)$ over the fixed alphabet A, and $INV(s_v, R_v)$ the set of all predicates over the free variables s_v, R_v with type $s_v : A^*, R_v : \mathbb{P}\, A$. Then we define a mapping

$$\iota : SPEC_{UF} \rightarrow INV(s_v, R_v)$$

by

$$\iota(S(s,R)) \equiv_{df} S(s_v, R_v)$$

The mapping ι just replaces the free variables s, R denoting trace and refusal in a CSP specification by the free variable s_v, R_v denoting *program variables* of sequential programs contained in ran ν. This replacement is well-defined since the traces s and variables s_v have the same type A^* and the refusals R and the variables R_v have the same type $\mathbb{P}\, A$.

Following this correspondence between CSP specifications $S(s,R)$ and predicates $\iota(S(s,R))$ over program variables s_v, R_v, we write

$$\begin{aligned}&\nu(P) \ \mathbf{sat} \ \iota(S(s,R)) \equiv_{df} \\ &\qquad (\forall\, \sigma, \sigma' : \Sigma(\nu(P));\ S' : \{E, S_L\} \bullet \\ &\qquad\qquad < \nu(P), \sigma > \longrightarrow^* < S', \sigma' > \Rightarrow (\sigma' \models (\forall\, U : \mathbb{P}\, R_v \bullet S(s_v, U))))\end{aligned}$$

where $\Sigma(\nu(P)), E, S_L$ are defined as in the previous section. Informally speaking, the assertion $\nu(P)$ **sat** $\iota(S(s,R))$ expresses that $\iota(S(s,R))$ is an invariant of the main loop of $\nu(P)$ and also holds in every final state, if $\nu(P)$ terminates.

3.4.5 Mapping Sequential Programs into the Failures Model

We will now define a mapping ξ from sequential nondeterministic processes in the range of ν into the untimed failures model $\mathcal{M}_{UF}$ of CSP.

$$\xi : SN \twoheadrightarrow \mathcal{M}_{UF}$$

$$\text{dom}\ \xi = \text{ran}\ \nu$$

For a sequential program $S \in$ ran ν we define

$$\begin{aligned}\xi(S) = \{&\sigma, \sigma' : \Sigma(S);\ S' : \{S_L, E\};\ U : \mathbb{P}\, A \mid \\ &\qquad < S, \sigma > \longrightarrow^* < S', \sigma' > \wedge\ U \subseteq \sigma'(R_v) \bullet (\sigma'(s_v), U)\}\end{aligned}$$

$\xi(S)$ defines the set of all pairs (u, U) such that u is the value of the program variable s_v and U a subset of the value of R_v, "recorded" at specific states σ' during an execution of S. The "recording" takes place at the beginning of each main loop cycle (program state S_L). If

S terminates (program state E), also the values of s_v and R_v in the termination state are added to $\xi(S)$.

ξ is obviously well-defined as a mapping into $A^* \leftrightarrow \mathbb{P}\,A$, the set of all relations between A^* and $\mathbb{P}\,A$. It still has to be shown that ran $\xi \subseteq \mathcal{M}_{UF}$. This is a consequence of the following lemma, exhibiting the relationship between ν, ξ and the semantic mapping $[\![\bullet]\!]_{UF}$[6]:

Lemma 11 *If $P \in CSP_{SNF}$ is a sequential CSP process in second normal form, then*

$$[\![P]\!]_{UF} = \xi \circ \nu(P)$$

Proof.
From Lemma 10 and the definition of ξ we get

$$\begin{aligned} Traces(P) &= \{u : A^* \mid \Sigma(\nu(P), u) \neq \emptyset \bullet u\} \\ &= \{\sigma, \sigma' : \Sigma(\nu(P));\; S' : \{E, S_L\} \mid <\nu(P), \sigma> \longrightarrow^* <S', \sigma'> \bullet\ \sigma'(s_v)\} \\ &= \{u : A^*;\; U : \mathbb{P}\,A \mid (u, U) \in \xi \circ \nu(P) \bullet u\} \end{aligned}$$

As a consequence it remains to be shown that

$$Ref(P/s) = \{U : \mathbb{P}\,A \mid (s, U) \in \xi \circ \nu(P) \bullet U\}$$

holds for all traces $s \in Traces(P)$.

To this end we calculate

$Ref(P/s) =$

[Lemma 8]

$$\begin{aligned} &\textstyle\bigcup_{\tau:\Sigma(P,s)}(\mathbf{if}\ \tau \models (\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i}) \\ &\qquad \mathbf{then}\ \mathbb{P}(A - \{\surd\}) \\ &\qquad \mathbf{else}\ \mathbb{P}(A - (\textstyle\bigcup_{(in,j):\{(c,i):\Gamma_{IN} \mid \tau\models B_{in,i}\}} \{\!|\ in\ |\!\} \cup \\ &\qquad\qquad \textstyle\bigcup_{(out,j):\{(c,i):\Gamma_{OUT} \mid \tau\models B_{out,i}\}} \{out.\tau(x_{out,j})\}))) = \end{aligned}$$

[Lemma 10]

$$\begin{aligned} &\textstyle\bigcup_{\sigma':\Sigma(\nu(P),s)}(\mathbf{if}\ \sigma' \models (\bigwedge_{(c,i)\in\Gamma_{IN}\cup\Gamma_{OUT}} \neg B_{c,i}) \\ &\qquad \mathbf{then}\ \mathbb{P}(A - \{\surd\}) \\ &\qquad \mathbf{else}\ \mathbb{P}(A - (\textstyle\bigcup_{(in,j):\{(c,i):\Gamma_{IN} \mid \sigma'\models B_{in,i}\}} \{\!|\ in\ |\!\} \cup \\ &\qquad\qquad \textstyle\bigcup_{(out,j):\{(c,i):\Gamma_{OUT} \mid \sigma'\models B_{out,i}\}} \{out.\sigma'(x_{out,j})\}))) = \end{aligned}$$

[definition of $R_v = r(\Gamma_{IN}, \Gamma_{OUT})$]

$$\textstyle\bigcup_{\sigma':\Sigma(\nu(P),s)}(\mathbb{P}\,\tau(R_v)) =$$

[definition of ξ]

$$\{U : \mathbb{P}\,A \mid (s, U) \in \xi \circ \nu(P) \bullet U\}$$

□

[6] Since we are considering processes P over a fixed alphabet A, $[\![P]\!]_{UF} = (A, F)$ is determined by the failures set $F : A^* \leftrightarrow \mathbb{P}\,A$ associated with P. Therefore we drop the A-component of $[\![\bullet]\!]_{UF}$.

3.4.6 Establishing Behavioural Properties of Normal Form Processes

We are now ready to prove the main theorem. It expresses the fact that the proof theory of Apt and Olderog [5] for sequential nondeterministic programs can be used to show that a CSP process in normal form satisfies a specification on traces and refusals.

Theorem 6 *If $P \in CSP_{SNF}$ is a sequential CSP process in normal form, interpreted in the untimed failures model $\mathcal{M}_{UF}$, then*

$$P \ \mathbf{sat}\ S(s,R) \quad \text{iff} \quad \nu(P)\ \mathbf{sat}\ \iota(S(s,R))$$

□

Proof. To establish the desired relationship, consider the following diagram:

$$\begin{tikzcd}
SPEC_{UF} \arrow[rr, "\iota"] \arrow[rr, phantom, "\text{triangle 1}"', shift right=4] \arrow[dr, leftrightarrow, "\mathbf{sat}"] \arrow[dd, leftrightarrow, "\mathbf{sat}"'] & & INF(s_v, R_v) \arrow[dl, leftrightarrow, "\mathbf{sat}"'] \arrow[dd, leftrightarrow, "\mathbf{sat}"] \\
{} \arrow[r, phantom, "\text{triangle 4}"] & \mathcal{M}_{UF} \arrow[r, phantom, "\text{triangle 2}"] & {} \\
CSP_{SNF} \arrow[ur, "{[\![\bullet]\!]_{UF}}"] \arrow[rr, "\nu"'] \arrow[rr, phantom, "\text{triangle 3}", shift left=4] & & \mathrm{ran}\ \nu \subseteq SN \arrow[ul, "\xi"']
\end{tikzcd}$$

The triangles 1,2,4 commute because of the definition of **sat** , triangle 3 commutes because of Lemma 11. Therefore the following transformations of predicates are valid:

$P\ \mathbf{sat}\ S(s,R)$

[definition of **sat** in untimed failures model]

$\Leftrightarrow (\forall (s,R) : [\![P]\!]_{UF} \bullet S(s,R))$

[Lemma 11]

$\Leftrightarrow (\forall (s,R) : \xi(\nu(P)) \bullet S(s,R))$

[definition of ξ]

$\Leftrightarrow (\forall (s,R) : \{\sigma, \sigma' : \Sigma(\nu(P));\ S' : \{S_L, E\};\ U : \mathbb{P}\, A \mid$

$$< \nu(P), \sigma > \longrightarrow^* < S', \sigma' > \wedge\ U \subseteq \sigma'(R_v) \bullet (\sigma'(s_v), U)\} \bullet S(s, R))$$

[set theory, semantics of nondeterministic sequential programs]

$$\Leftrightarrow (\forall\, \sigma, \sigma' : \Sigma(\nu(P));\ S' : \{S_L, E\} \bullet$$
$$< \nu(P), \sigma > \longrightarrow^* < S', \sigma' > \Rightarrow (\sigma' \models (\forall\, U : \mathbb{P}\, R_v \bullet S(s_v, U))))$$

[definition of **sat** for nondeterministic sequential programs]

$$\Leftrightarrow \nu(P)\ \mathbf{sat}\ \iota(S(s, R))$$

This completes the proof of Theorem 6.
□

3.5 Discussion and Future Work

In this Chapter we have presented a case study, describing the design and verification of a fault-tolerant dual computer system. The case study was presented in the formal framework introduced in Chapter 2. This time we demonstrated on system level how lifeness properties can be verified by means of reasoning about behavioural specifications in the failures-divergence model, without using the abstract interpretation techniques applied for the case study in Chapter 2. Another new proof technique was introduced, allowing to verify behavioural properties of explicit sequential CSP processes in the failures model by means of reasoning about sequential nondeterministic programs. The proof method was formally justified by constructing a mapping from normal form CSP processes into sequential nondeterministic programs and proving that behavioural specifications of the former could be equivalently expressed as invariants of the latter.

With an explicit process representation at hand which is known to be consistent with the implicit specifications developed during the preceding design stages we have two possibilities to reach implementable processes:

- If the sequential CSP process representation is sufficiently "close" to the executable language of the target system, source code can be produced by means of syntactic transformations. For example, the processes $ABFTX$, $ABFRC$ introduced in Section 3.3.2 could be implemented in OCCAM in a rather straightforward way. Such techniques have been investigated in the ProCoS project [10] with the objective to generate OCCAM code for the Inmos Transputer [46]. In the UniForM project [88] a similar approach will be undertaken to generate PLC code from normalised CSP process representations.

- If the sequential CSP process P still represents rather an abstract view on the system, possibly not yet showing all the relevant implementation details, further explicit CSP refinements $Q_1, Q_2, Q_3, \ldots$ may be produced and verified by means of model checking with FDR [27]. Such refinements can introduce new parallel processes showing additional channels and other implementation details, so that the refinement condition would be $P \sqsubseteq\ Q_i \setminus L(Q_i)$, where $L(Q_i)$ denotes the new local events of Q_i introduced in the refinement step. This technique has been applied in the verification of the components of the dual computer system, where the associated higher-level sequential CSP process only served as an auxiliary component "closing the gap" between behavioural specifications and explicit processes.

The relation between CSP processes and nondeterministic sequential programs described above is not the only one: It is well-known that also $\|$- and $|||$-combinations of sequential normal form processes can be transformed into equivalent sequential programs. This *Sequentialisation Theorem* has been proven by Apt and Olderog for their operational CSP semantics [5, p. 336]. For the denotational CSP semantics a proof is currently worked out in the UniForM project, as a generalisation of the proof presented above. However, this theorem is not essential in the development framework for dependable system described in Chapter 2: Having established the behavioural properties of isolated sequential processes with the help of Theorem 6, the properties of their $\|$- or $|||$-combination can be derived applying the laws of the compositional CSP proof theory for these operators.

It should be noted that from today's point of view, the fault-tolerant dual computer system should be regarded as a "toy-example". It would have been possible to verify the full system mechanically in a single step, using the FDR model checker. Nevertheless the mechanisms of the hot standby design show the crucial properties necessary for a "real" fault-tolerant server implementation, like the one described in [73]. Moreover, I am convinced that the method to apply different verification formalisms, as demonstrated in the case study, is very helpful for the development of "real-world" systems: It allows to benefit from the theoretical results and the tool support available in one formalism (in our case, sequential nondeterministic programs) to establish properties about objects defined in another (in our case, CSP processes).

Besides using combinations of different techniques, a promising approach to facilitate the verification process is to use a "library" of verified "generic" specifications, so that new concrete solutions could be verified to be "instantiations" of certain specifications of the library. Such techniques help to avoid verifying each new development from the most abstract level down to the detailed design. A first effort in that direction has been undertaken by the author in [76], where a modified version of the dual computer was proven to be consistent with the system presented above, using the algebraic proof theory of CSP. For a successful application of such an approach it is essential to investigate the basic *paradigms* of fault-tolerance, in order to decide which abstract specifications are the best candidates for a generic library. An initial investigation of these paradigms has been performed in the NWO project *"Fault Tolerance: Paradigms, Models, Logics, Construction"* [103]. The hot standby concept described above seems to represent such a paradigm. Further candidates for generic specifications are *stable storage* [103], n-modular redundant computers with *voting mechanisms* [104] and fault-tolerant *communication protocols* like the one used in Chapter 2.