Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Formal Methods and the Development of Dependable Systems

Jan Peleska

e-mail: jp@informatik.uni-bremen.de

Bericht Nr. 9612
Dezember 1996

Dieser Bericht enthält die Habilitationsschrift des Verfassers.

Referent: Prof. Dr. Willem-Paul de Roever
Koreferenten: Prof. Dr. Manfred Broy, Prof. Dr. Mathai Joseph, Prof. Dr. Hans Langmaack, Dr. Tech. Anders P. Ravn

Preface

This *Habilitationsschrift*[1] focuses on methods for the development of dependable software-based systems. It summarises, discusses and extends my publications cited in the bibliography, which reflect the efforts and experiences gained in this field during the last decade, working as a software engineer, project leader and manager at *Philips GmbH*, *Deutsche System-Technik GmbH* and as a consultant for various other companies.

The applications to be discussed as examples will mostly be chosen from the field of non-military information and control systems. Such a specialisation appears to be necessary, because the type of application influences the objectives to be met by dependability mechanisms. For example, in dependable and secure military applications (see [50]), the aspect of confidentiality (mandatory access control, covert channels etc.) plays a much more important rôle than for a railway control system. As a consequence, different application areas lead to different approaches with respect to system design, development techniques and underlying specification and design methods.

The methodological framework used is based on *Formal Methods*. I would like to point out that though my university education was rather theoretical (see [72]), the motivation to use theory for building software was purely motivated by the fact that the informal heuristics applied for constructive or analytic software quality assurance are completely insufficient when applied to systems where correctness of software really matters. The key ideas described in the chapters to follow might be summarised by three statements reflecting my conviction how the software crisis should be tackled in the future:

- The complexity of today's applications can only be managed by applying *a combination of methods*, each of them specialised to support specific development steps in an optimised way during the system development process.
- The application of formal methods should be supported by *development standards*, i. e., explanations or "recipes" showing how to apply the methods in the most efficient way to a specific type of development task. Indeed, tool support and development standards may be regarded as essential for the success of formal methods in an industrial context.
- The application of formal methods for the development of dependable systems will only become cost-effective if the degree of *re-usability* is increased by means of re-usable (*generic*) specifications, re-usable proofs, code and even re-usable development processes.

As a consequence, the objective of this *Habilitationsschrift* is not to introduce new specification languages, proof techniques *et cetera*, but to show how new combinations of existing methods can be applied more efficiently to solve problems in the field of dependable systems.

[1] *Postdoctoral thesis*, required in Germany for the qualification of a university lecturer; since the English language does not provide a proper counterpart for this term, we keep the German expression instead of using an English circumscription.

The "architecture" of this work is depicted in Figure 0.1. In the introductory Chapter 1 some basic definitions used in the context of dependability are presented and discussed, since the "invasion" of software into this field has forced specialists to re-interpret important terms previously used only in the context of mechanical and electrical engineering. Moreover, an example is given which reflects the complexity to be encountered when trying to combine different dependability mechanisms (e. g. a fault-tolerant protocol together with a security mechanism) in order to "add up" the benefits of these mechanisms in one system. In Chapter 2 I will describe an approach for the systematic formal development of dependable systems. This approach offers a design technique systematically dealing with situations where not just one threat to system dependability but a collection of threats has to be taken into account. At the same time it is embedded in a general development framework that has proved to be practical for large projects and is accepted by (at least the more enlightened species of) today's software engineers in industry. This embedding associates a theoretic foundation with an informal development standard, using selected formal methods to produce the contents of the documents pre-defined in the standard. The example of Chapter 1 is used to demonstrate how the approach can be put into practice. This work has been mainly motivated by my cooperation with ELPRO LET GmbH and the University of Oldenburg in the field of distributed railway interlocking systems.

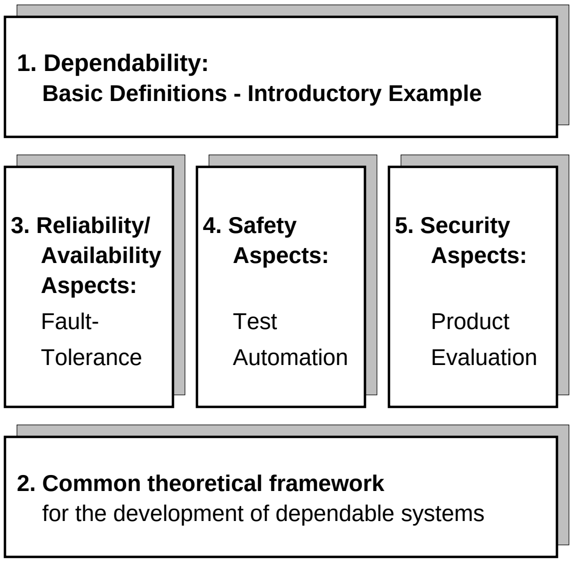

Figure 0.1: "Architecture" of the *Habilitationsschrift*.

Chapters 3 to 5 focus on specific aspects and applications in the field of dependable systems: reliability/availability, safety and security. In Chapter 3 the formal specification and verification of a dual computer system is described, where the reliability of the system is ensured by means of a fault-tolerance mechanism. The underlying design and verification concepts have been worked out during my time with Philips, where I was confronted with the development of a fault-tolerant database server. In Chapter 4 a concept for the automated test of safety-critical embedded systems will be introduced. The concept and the corresponding tool support have been developed for and applied in the field of railway interlocking systems for small private railways or tramways. For these systems, the degree of complexity is considerably lower than in the case of large-scale railway networks. Furthermore, safety aspects play the dominant rôle, while reliability and availability are subordinate aspects. Therefore we can achieve a much higher degree of automation for the test and verification process than for the case of dependable systems in general. In Chapter 5 security aspects of dependable systems are investigated. Here we concentrate on the evaluation of commercially available IT security products. The objective of this evaluation is to certify in a trustworthy way that the system is capable to protect its users against the specified types of security attacks. This work has been initiated while I was manager of a department at DST that was specialised on developing dependable systems. Associated with the department was an evaluation laboratory which was accredited at the *Bundesamt für Sicherheit in der Informationstechnik BSI*, the German authority for the certification of IT security products.

You may have noticed that I did not use the Preface to give a “sales talk” about how much we are threatened by software of insufficient quality and how only formal methods can deliver us from this evil. Firstly, I assume that this *Habilitationsschrift* will only be read by experts anyway, and either you will already be a member of the formal methods community or you will be too much of an expert to be convinced by my arguments. Secondly, I have the impression that the common understanding about the *status quo* of the software crisis, what formal methods can do about it and where the limitations of formal methods lie has improved very quickly during the last five years. This even holds for hard-boiled software practitioners to be encountered in software industry. If you still feel like reading some motivating thoughts about all this, I am sure the articles of Gibbs [31], Heisel and Weber-Wulff [38] and the Hamer-Hörcher-Peleska formal methods primer [79] will satisfy your needs.

Acknowledgements While writing (and re-writing) this *Habilitationsschrift* I have been very aware of how much we depend on the support of other people, whenever a sincere effort shall meet its goals. I would like to express my gratitude to all those who have helped me during the completion of this work, for their advice, both technical and personal, their encouragement, their patience and sometimes simply for the splendid time we've had together:

Alexander Baer, Walter Benz, Dines Bjørner, Chris Brink, Manfred Broy, Bettina and Karl-Heinz Buth, Rachel Cardell-Oliver, Di Dixon, Manfred Endreß, Ute Hamer, Ilse and Heinz-Otto Hamer, Ute Hammerich, Rait Harnett, Maritta Heisel, Mike Hinchey, Tony Hoare, Hans-Martin Hörcher, Kees Huizing, Mathai Joseph, Dagmar König, Bernd Krieg-Brückner, Hans Langmaack, Maureen Le Sar, Erich Mikk, Ernst-Rüdiger Olderog, Astrid Peleska, Erika and Viktor Peleska, Carsta Petersohn, Erhard Pompe, Anders P. Ravn, Hans Rischel, Corinne and Willem-Paul de Roever, Fred Schindler, Michel Schrönen, Michael Siegel, Mike Spivey, Änne Straßner, Jan Vytopil, Jim Woodcock, Margarete Worm

Contents

List of Figures

1. Introduction

1.1 Dependability — Basic Definitions and Concepts

Throughout this work, we will follow Laprie's terminology [56, 57, 58] with respect to dependability and related terms.

Generally speaking, *dependability is the capability of a system to deliver the specified application services during its period of operation.* This definition emphasizes two aspects that influence the design, the implementation and our reasoning about system "correctness" in a crucial way:

- Dependability does *not* forbid the occurrence of failures in general. Instead, it requires that the *application service*, i. e., the functionality required by the end user of the system is delivered as specified.
- Delivery of service is only required to the extent *covered by the specification*. This is a very reasonable requirement – otherwise the supplier would never have a chance to prove that the system has been completed and the customer's requirements have been implemented. On the other hand, the approach is only useful, if the technical contents of a specification document reflecting how a system *will* behave according to the developers' understanding is consistent with the end user's intuitive understanding of how the system *should* behave. As a consequence, the specification phase has become the most critical phase of the whole system development life cycle.

Laprie identified four attributes which characterise the dependability of a system: (1) A *safe* system cannot assume states that are regarded as "catastrophic" from the point of view of the application. This means that the system will only perform transitions into states satisfying the specified invariants, perform calculations that are correct with respect to the specification and output data fulfilling the desired integrity constraints. Safety does not guarantee that a desired calculation and the corresponding output will always be produced. This aspect is covered by the following two attributes: (2) *Reliability* is a characteristic specifying the probability that a system will deliver its service for a given period of time[1]. (3) *Availability* is a measure reflecting the probability that the system will be available at a certain point in time. (4) Finally, *Security* reflects the capability of the system to protect the application against damage arising from accidental or malicious human interaction. All dependability attributes refer to the specified application, and this is the very premise for the notion of *fault-tolerance*: A dependable system may be subject to various *internal* defects, as long as these problems do not affect the application behaviour.

For the security attribute, a further characterisation has been provided: According to the standards [17, 28, 29, 47, 111] the notion of *Information Technology (IT) Security* is defined

[1] For example, the *mean time between failure (MTBF)* is a reliability measure.

by the attributes *Confidentiality* (prevention of unauthorised disclosure of information), *Integrity* (prevention of unauthorised modification of information) and *Availability* (prevention of unauthorised withholding of information or resources)[2]. IT security focuses on the damage that can be caused by humans. In [17, p. 3] this is made explicit by defining IT security as *"...protection of information ... by countering threats to that information arising from human activities whether malicious or otherwise."* There is at least one good reason to distinguish between threats arising from technical deficiencies and those arising from human activities: In many cases technical faults can be considered as accidental events and therefore adequately modelled by statistical approaches. These modelling techniques will fail, however, when being applied to the behaviour of malicious human intruders, who can try to cause damage in a systematic way.

Since the definition of dependability is intended to be applicable to a wide spectrum of systems and the aspects of dependability are intended to be describable by a wide range of formalisms, Laprie's definitions are of informal nature. It is useful to relate them to the formal notions of *safety* and *liveness*. Recall that in the context of formal methods a specification item S is a *safety property*, if any sequence of events or transitions etc. violating S contains a prefix all of whose infinite extensions violate S. A specification item L is called a *liveness property*, if any arbitrary finite sequence of events can be extended to an infinite sequence satisfying L [62, p. 303]. Obviously, the dependability attribute *safety* is also a safety property in the sense of formal methods. The situation is less obvious for *availability*: In the context of hard real-time systems, it is best interpreted as another safety property, because the availability of a service is then interpreted as *"available within n time units"*, and an assertion saying that the service will "finally" be available is not helpful at all. For less time-critical applications, however, the notion of availability might be associated with *"the service will not be blocked forever"* or *"the service will be activated sufficiently often"*, and these are liveness properties. As a consequence, the formalisation of the availability definition depends on the specific application. *Reliability* is only indirectly related to safety and liveness: Intuitively, this dependability attribute describes the possibilities of the system to "switch" from one specified behaviour to another behaviour which is regarded as exceptional. Of course, both behaviour specifications can be decomposed into safety and liveness properties. The *confidentiality* and *integrity* aspect of security is a safety property. For the *availability* aspect of security the same considerations apply as for availability in general.

The notion of security is a "new" dependability attribute, due to Laprie. For example, in the definition given 1985 in [7], security is not yet mentioned, and even in a fairly recent survey [107] on real-time computing the authors state that *...it is not very important in a real-time operating system to provide extensive support for ... security*. I think that this fact is characteristic for the type of systems designers had – and sometimes still have – in mind when coining the notion of dependability in the context of computer science for the first time. The major threats to dependability were hardware and software errors and external physical events. Threats caused by humans were thought to be similar to natural disasters like acts of vandalism. The more "subtle" aspects of security were not considered as

[2]Obviously, these attributes introduce redundancy in the dependability definition, since availability has already been introduced as a main attribute of dependability and integrity can be regarded as a safety aspect. These issues have been discussed by Jonsson and Olovsson [53]. However, it is not our objective to try and improve existing definitions, but to find trustworthy ways how to build systems according to given dependability requirements.

relevant in the early eighties. On the other hand, it has always been an obvious requirement that a dependable system should also be secure because otherwise, a human attacker might systematically damage the integrated dependability mechanisms. However, in the "early times" of dependable computerised systems, security issues could often be ignored *in the software design*, because the operational environment was *automatically secure*, i. e. did not possess any interfaces accessible to a human attacker. In recent years, a growing demand for *open* and/or *distributed* dependable systems became apparent. Obviously, these new key attributes force us to treat security as a global system issue which has to be covered by protective mechanisms both in the operational environment *and* in the software. Examples for such projects are given in Chapter 2 and in [78]. From my experience it is interesting to note that the demand for open, distributed dependable systems came from two sides:

- In the "classical" application domains of dependable systems openness and distribution have often been introduced to reduce costs by providing a system architecture that can be customised and extended more easily.
- In less critical applications dependability became an important aspect *a posteriori*, in order to reduce costs by increasing the operational availability and to avoid loss or corruption of valuable data. These systems were often open and distributed by their very nature, and the new dependability features had to be fitted into this existing architectural framework.

I will use the term *classical dependability* when referring to safety, reliability and availability only.

To illustrate the design tasks to be tackled for the development of dependable systems the classification of system behaviours depicted in Figure 1.1 will be used. In absence of any undesired (internal or external) events, the system executes[3] *normal behaviour*. In presence of undesired events the system may perform an execution possibly differing from that of the undisturbed system. Now there are two possibilities: In the first case, an undesired event impairs the functionality of the system in a way that can be tolerated or might even be unobservable in the application layer. In this case the system shows *exceptional behaviour*. If the system performs an execution showing normal and/or exceptional behaviour, this is summarised as *acceptable behaviour*. In the second case, the occurrence of the undesired event corrupts the application functionality in a way that cannot be tolerated. This is called *catastrophic behaviour*. Although the above classification has been introduced in [59] to describe concepts of fault-tolerance only, it fits very well to deal with dependability as a whole.

In order to prevent undesired events from leading to catastrophic system behaviour, system designers must anticipate the possibility of their occurrence and incorporate protective mechanisms in their development concepts. The potential occurrence of any undesired event will be called a *Threat* or a *Fault Hypothesis*. If it is necessary to distinguish between aspects of classical dependability and security, we will denote the fault hypotheses anticipating human interactions as *Security Threats*. The actual occurrence of any undesired event is called a

[3]Depending on the formalism used, an execution may be a modelled as a *trace of events*, hiding internal state information, or as a sequence of *state transitions* triggered by events. For hybrid systems, the execution is modelled as some type of *trajectory*, traversing an appropriate space-time model spanned by state and time parameters.

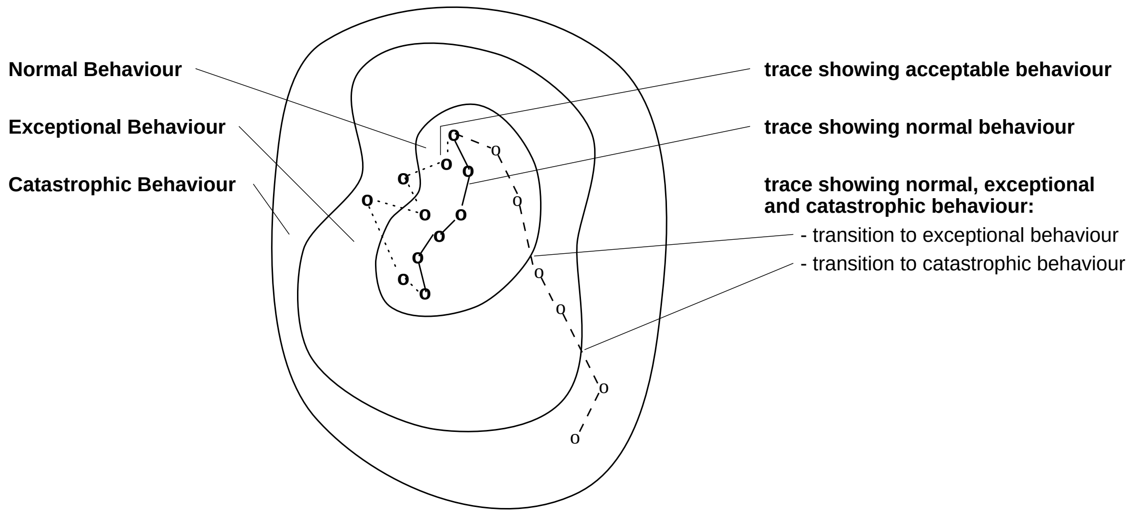

Figure 1.1: Types of system behaviour.

Fault. Faults caused by security threats are termed *Human-Made Faults*; we will also use the more suggestive term *Security Attack* and summarise human-made and other faults as *Exceptions*. The process of anticipating exceptions will be denoted as *Threat Analysis*. If an exception causes a transition into an undesired system state, this state is called an *Error*. If due to the occurrence of exceptions and errors a specified service to be performed by a (sub-)system cannot execute as required, this is called a *Failure*. In the context of real-time systems it is often necessary to distinguish *Value Failures*, where incorrect data is produced or processed by the system from *Timing Failures*, where correct data appears at the wrong time.

The means to prevent faults, errors and failures are *Fault Avoidance* (building "perfect" hardware and software components by means of special construction techniques), *Fault Tolerance* (use of techniques allowing the continuation of service in presence of faults), *Verification* (detection and removal of errors by means of analytic techniques) and *Error Forecasting* (analysis of the impact of specific errors and of the probability for their occurrence). The objects implemented to prevent the undesired consequences of security attacks are called *Security Mechanisms*.

To model the system behaviour in presence of exceptions, two main techniques have been used by different authors:

- *Explicit specification,* for example by specifying the system and the impact of exceptions in terms of a process algebra. This method will be demonstrated in the next section. Further applications are presented in Chapter 3.
- *Behavioural specifications:*[4] The system behaviour and the impact of anticipated exceptions is described by predicates on traces of events or executions of a transition system [70, 104].

[4] Following Jones [52], we also use the term *implicit* specifications.

1.2 Combination of Dependability Mechanisms — Case Study Part I

While many publications investigate isolated dependability aspects[5], relatively few of them are concerned with combinations of these aspects. I am convinced, however, that it is just this combination that introduces a new degree of complexity in the field of dependable systems, because

- it can be dangerous to combine two dependability mechanisms in a naive way and hope that the resulting system will automatically possess the strength of both mechanisms,
- the design of correctly cooperating dependability mechanisms is likely to differ strongly from solutions where only one isolated mechanism is required.

To support these theses, we will now analyse a case study focusing on the development of a network protocol which is dependable in the presence of both unreliable and insecure transmission media (Figure 1.2). The case study is a revised version of the example worked out in [78]. In the remaining part of this chapter it will serve to illustrate the complexity to be expected when more than one dependability threat is involved: two mechanisms correctly defending the system against the reliability threat and the security threat alone, respectively, will result in an *unreliable* protocol when combined in a "naive" way. This failure to construct "globally" dependable mechanisms from isolated solutions in an intuitive way will motivate the systematic approach for the development of dependable systems as introduced in Chapter 2.

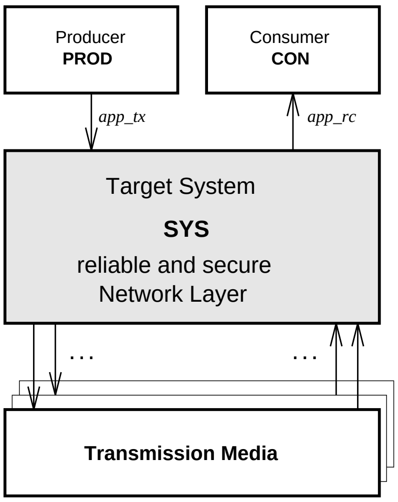

Figure 1.2: Context of the target system (network layer) and its environment.

In the examples to follow the number of transmission media and their various deficiencies will vary in order to illustrate different aspects of dependability. The dependability requirement will always remain the same, however:

- Construct *SYS* in such a way that the resulting communication service will act for producer *PROD* and consumer *CON* as a FIFO buffer without loss of data or corruption of messages.

[5] Rushby [102] has collected a comprehensive bibliography.

I will use CSP (*Communicating Sequential Processes*) as introduced by Hoare [44] to specify and reason about system behaviour. Correctness properties will be expressed by means of *failures-divergence refinement* relating the CSP process expressing correct behaviour to the protocol solutions proposed in the examples. The refinement steps presented have been mechanically checked using the FDR tool [27]. FDR allows to verify refinement properties by means of complete model checking for CSP systems consisting of cooperating finite-state sequential processes. The specification by explicit processes only serves for illustration purposes in these examples. In Chapter 2, a behavioural specification style will be used which is more adequate for the abstract description of large systems.

The informal dependability requirement can be specified by relating an implementation X to the explicit CSP process[6] *FIFO* defined as

$$\alpha(\mathit{FIFO}) = \{|\ \mathit{app\_tx}, \mathit{app\_rc}\ |\}$$

$$\mathit{FIFO} = \mathit{BUFF}(\langle\ \rangle)$$

$$\begin{aligned}\mathit{BUFF}(s) = \ & (\#s < N)\&\mathit{app\_tx}?x \rightarrow \mathit{BUFF}(s ^\frown \langle x \rangle) \\ & \Box \\ & (\#s > 0)\&\mathit{app\_rc}!(\mathit{head}(s)) \rightarrow \mathit{BUFF}(\mathit{tail}(s))\end{aligned}$$

which represents a FIFO buffer of finite capacity N: Any X consisting of the transmission media and the protocol layer SYS used in the implementation has to satisfy

$$\mathit{FIFO} \ \sqsubseteq_{FD} \ X \setminus (\alpha(X) - \{|\ \mathit{app\_tx}, \mathit{app\_rc}\ |\})$$

where $P \ \sqsubseteq_{FD} \ Q$ means that P is refined by Q in the failures-divergence model of CSP.

Example 1.1 (Communication disturbed by unreliable media) We first consider the case where only unreliable transmission media are available, behaving as follows:

1. **Fault hypothesis:** The medium may lose messages. The number of consecutive input messages that the medium may lose is bounded by a number $\mathit{maxLoss} \geq 0$.
2. The medium does not change the values of messages delivered to the consumer.
3. The medium delivers messages in the order of the corresponding inputs.

In order to design a fault-tolerant network layer taking into account the above fault hypothesis, the architecture shown in Figure 1.3 is used (the "lightning" symbols indicating the possibility of data losses). $M1, M1ACK$ are two instances of the unreliable medium. Their behaviour according to the above assumptions can be expressed by the following CSP processes:

[6] We use *communication guards*: in an expression $b\&c$, the communication via channel c is refused by the process, if the Boolean expression b evaluates to *false*. $\{|\ c, d, \ldots\ |\}$ denotes the set of channel events $\{c.x_1, c.x_2, \ldots, d.y_1, d.y_2, \ldots\}$, x_i and y_j are values of the channel alphabets of c and d, respectively.

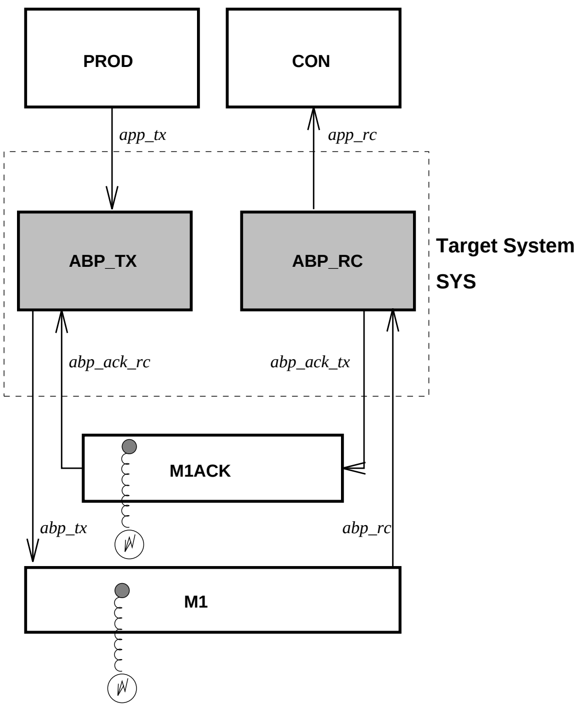

Figure 1.3: Unreliable transmission media and alternating bit protocol implemented by ABP_TX, ABP_RC.

$$
\begin{aligned}
&M1 = M1(maxLoss)\\[1em]
&M1(n) = abp\_tx?(b,x) \rightarrow\\
&\qquad\qquad (\mathbf{if}\ n = 0\ \mathbf{then}\\
&\qquad\qquad\qquad (abp\_rc!(b,x) \rightarrow M1(maxLoss))\\
&\qquad\qquad \mathbf{else}\\
&\qquad\qquad\qquad (abp\_rc!(b,x) \rightarrow M1(maxLoss)\\
&\qquad\qquad\qquad \sqcap\\
&\qquad\qquad\qquad M1(n-1)))
\end{aligned}
$$

$$
\begin{aligned}
&M1ACK = M1ACK(maxLoss)\\[1em]
&M1ACK(n) = abp\_ack\_tx?b \rightarrow\\
&\qquad\qquad (\mathbf{if}\ n = 0\ \mathbf{then}\\
&\qquad\qquad\qquad (abp\_ack\_rc!b \rightarrow M1ACK(maxLoss))\\
&\qquad\qquad \mathbf{else}\\
&\qquad\qquad\qquad (abp\_ack\_rc!b \rightarrow M1ACK(maxLoss)\\
&\qquad\qquad\qquad \sqcap\\
&\qquad\qquad\qquad M1ACK(n-1)))
\end{aligned}
$$

To construct a reliable network layer on top of $M1, M1ACK$, we use the well-known *alternating bit protocol* (e. g., see [64, 104] for two alternative versions of the protocol) consisting of a transmitter ABP_TX and a receiver ABP_RC with $SYS = (ABP\_TX \mid\mid\mid ABP\_RC)$. ABP_TX receives data from the producer, attaches a bit value alternating with each new message accepted on app_tx and sends it via $M1$ to ABP_RC. The receiver passes data via app_rc to the consumer, if the attached bit has the expected value in the alternating sequence. In any case ABP_RC sends the bit value received back to ABP_TX via medium $M1ACK$ to acknowledge reception. In our ABP solution ABP_TX will continuously transmit the actual $(bit, data)$-package until it receives an acknowledgement with the same bit value. The corresponding CSP processes are

$$ABP\_TX = ABP\_TX(1)$$

$$\begin{aligned}
&ABP_TX(bit) = \\
&\qquad app_tx?x \rightarrow ATX((1 - bit), x) \\
&\qquad \square \\
&\qquad abp_ack_rc?b \rightarrow ABP_TX(bit)
\end{aligned}$$

$$\begin{aligned}
&ATX(bit, v) = \\
&\qquad abp_tx!(bit, v) \rightarrow ATX(bit, v) \\
&\qquad \square \\
&\qquad abp_ack_rc?b \rightarrow \\
&\qquad\quad (\textbf{if } (b = bit) \textbf{ then } ABP_TX(bit) \textbf{ else } ATX(bit, v))
\end{aligned}$$

$$ABP\_RC = ABP\_RC(0)$$

$$\begin{aligned}
&ABP_RC(bit) = \\
&\qquad abp_rc?(b, x) \rightarrow \\
&\qquad\quad (\textbf{if } (b = bit) \\
&\qquad\quad\ \textbf{then } (app_rc!x \rightarrow abp_ack_tx!b \rightarrow ABP_RC(1 - bit)) \\
&\qquad\quad\ \textbf{else } (abp_ack_tx!b \rightarrow ABP_RC(bit)))
\end{aligned}$$

The transmission media and SYS cooperate according to

$$ABP = (M1 \,|||\, M1ACK) \,||\, (ABP\_TX \,|||\, ABP\_RC)$$

To prove that ABP is an acceptable implementation of our ideal transmission medium, we have to show that

$$FIFO \sqsubseteq_{FD} ABP \setminus \{|\ abp\_tx, abp\_rc, abp\_ack\_tx, abp\_ack\_rc\ |\}$$

which can be proven for finite $maxLoss$ and buffer capacity $N = 1$ using the FDR model checker.
□

Example 1.2 (Communication disturbed by insecure media) Let us now consider a transmission medium that is physically reliable but suffers from attacks by a malicious agent X. In the field of communication protocols, the most important types of attack have been classified as follows (see [9]): *(1) Eavesdropping:* Messages sent from $PROD$ to CON are properly received by CON, but X can also receive (some of) them. *(2) Blocking:* X can receive messages intended for CON and prevent them from being delivered to CON.

(3) Delay: X can delay the transmission from $PROD$ to CON. *(4) Masquerading:* X can send a fake message to CON that looks as if it has been sent by $PROD$. *(5) Replay:* X can send an old message that has already been sent from $PROD$ to CON once more at a later point in time. *(6) Tampering:* X can alter a message while it is transmitted from $PROD$ to CON, so CON receives modified data.

In this example, let the following behaviour be defined for the transmission medium:

1. **Security threat:** X can perform eavesdropping and tamper with data sent from $PROD$ to CON. Making use of tampering, X can also replay a message by "copying" an old message onto a new package. X cannot block or delay messages or fake the identity of $PROD$. X can modify at most *maxModified* ≥ 0 messages in a row.
2. The medium delivers messages in the order of the corresponding inputs.
3. The medium does not lose messages.
4. While transmission of user data can be tampered with, there exists a separate reliable and secure communication channel for the transmission of control data.

The last condition may seem slightly artificial. I have added it in order to construct an extremely simple security protocol that works properly in this example but fails when combined with a fault-tolerance mechanism, as will be shown in the next example.

To defend the application layer against the insecure transmission medium $M2$ the architecture depicted in Figure 1.4 is chosen.

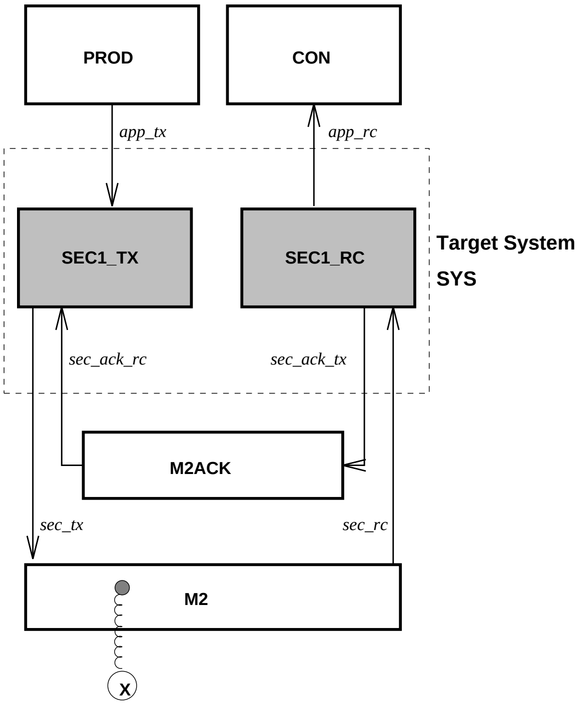

Figure 1.4: Security layer $SEC1\_TX, SEC1\_RC$ to protect the application layer against eavesdropping and tampering on $M2$ by agent X.

The insecure behaviour of $M2$ may be modelled as follows:

$$
\begin{aligned}
&M2 = M2(maxModified)\\[1em]
&M2(n) = sec\_tx?x \to\\
&\qquad (\textbf{if } n = 0 \textbf{ then}\\
&\qquad\qquad (sec\_rc!x \to M2(maxModified))\\
&\qquad\ \textbf{else}\\
&\qquad\qquad (sec\_rc!x \to M2(maxModified)\\
&\qquad\qquad \sqcap\\
&\qquad\qquad sec\_rc!modify(x) \to M2(n-1)))
\end{aligned}
$$

Here output $sec\_rc!modify(x)$ represents the case where X has tampered with input $sec\_tx.x$. Counter n prevents $M2$ from sending more than $maxModified$ corrupted outputs in a row. $M2ACK$ is assumed to be dependable and acts as a one-place buffer:

$$M2ACK = sec\_ack\_tx?b \to sec\_ack\_rc!b \to M2ACK$$

To meet the security threat represented by $M2$, the security layer implemented in SYS will use encryption techniques to detect altered or replayed messages and to prevent X from reading them. This technique will be explained in more detail in Chapter 2; for now it is sufficient that we may assume the existence of an encryption procedure ϵ and an detection algorithm which signals $isModified(x) = true$ if a package x has been tampered with. By ϵ^- we denote the corresponding decoding procedure. The security layer may now be specified as

$$
\begin{aligned}
&SEC1\_TX = app\_tx?x \to sec\_tx!\epsilon(x) \to S1TX(\epsilon(x))\\[1em]
&S1TX(x) = sec\_ack\_rc.Modified \to sec\_tx!x \to S1TX(x)\\
&\qquad\qquad \square\\
&\qquad\qquad sec\_ack\_rc.Valid \to SEC1\_TX\\[1em]
&SEC1\_RC = sec\_rc?y \to\\
&\qquad (\textbf{if } isModified(y)\\
&\qquad\ \textbf{then } (sec\_ack\_tx!Modified \to SEC1\_RC))\\
&\qquad\ \textbf{else } (app\_rc!\epsilon^-(y) \ \to sec\_ack\_tx!Valid \to SEC1\_RC)
\end{aligned}
$$

Process $SEC1\_TX$ accepts inputs from the application layer via channel app_tx. It enciphers the message by means of ϵ and passes the result to $M2$. As long as X does not know the key to decipher the message, it can only corrupt the data at random or copy an old package over the new data block. But this will be detected by $SEC1\_RC$ using the $isModified$ algorithm on the input y received on sec_rc. If the message has been modified or is a replay of an old one, $SEC1\_RC$ will send a control message $Modified$ via $M2ACK$ to $SEC1\_TX$ and the latter will re-transmit the enciphered message. Since according to the assumptions neither $M2$ nor $M2ACK$ lose messages and agent X can only modify, but not interrupt the communication on $M2$, $SEC1\_TX$ can be sure to receive a response from $SEC1\_RC$ for each message transmitted. It is therefore not necessary to watch for timeouts while waiting for the control message. If the message has been received by $SEC1\_RC$ without previous modification, the deciphered message contents will be passed on to CON, and control message *Valid* is sent back to $SEC1\_TX$. After that $SEC1\_TX$ is ready to accept new data from $PROD$.

The complete secure implementation cooperates according to

$$SEC = (M2 \mid\mid\mid M2ACK) \parallel (SEC1\_TX \mid\mid\mid SEC1\_RC)$$

and again it may be shown using the FDR tool that

$$FIFO \sqsubseteq_{FD} SEC \setminus \{\mid sec\_tx, sec\_rc, sec\_ack\_tx, sec\_ack\_rc \mid\}$$

for fixed values *maxModified* and buffer capacity $N = 1$. The example does not show how to initiate communication and exchange keys for enciphering/deciphering between $SEC1\_TX$ and $SEC1\_RC$. Possible solution for such identification, authentication and on-line key distribution procedures have been described in [9, 19].
□

Example 1.3 (Naive combination of protocols in presence of more than one threat) Now what happens if we are confronted with a transmission medium $M3$ which is both unreliable in the sense of the first example and insecure in the sense of the second? It may be tempting to try and combine the existing solutions in an architecture as shown in Figure 1.5. $SEC1\_TX, SEC1\_RC, M2ACK$ are chosen as in the previous example. $ABP1\_TX, ABP1\_RC$ implement an alternating bit protocol on top of $M3, M1ACK$. They are copies of ABP_TX/RC above, with channels app_tx, app_rc renamed to sec_tx, sec_rc. This seems to work at first sight, because the ABP layer makes up for lost messages and the security layer can deal with corrupted data.

We assume that $M1ACK$ behaves as in Example 1, but $M3$ now combines the fault hypotheses and security threats of the previous examples according to

$$
\begin{array}{l}
M3(n) = abp\_tx?(bit, x) \rightarrow \\
\qquad\qquad (\textbf{if } n = 0 \textbf{ then} \\
\qquad\qquad\quad (abp\_rc!(bit, x) \rightarrow M3(maxModifiedOrLost)) \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\quad (abp\_rc!(bit, x) \rightarrow M3(maxModifiedOrLost) \\
\qquad\qquad\quad \sqcap \\
\qquad\qquad\quad M3(n-1) \\
\qquad\qquad\quad \sqcap \\
\qquad\qquad\quad abp\_rc!(bit, modify(x)) \rightarrow M3(n-1) \\
\qquad\qquad\quad \sqcap \\
\qquad\qquad\quad abp\_rc!((1-bit), x) \rightarrow M3(n-1) \\
\qquad\qquad\quad \sqcap \\
\qquad\qquad\quad abp\_rc!((1-bit), modify(x)) \rightarrow M3(n-1)))
\end{array}
$$

And this is the reason why this combination of mechanisms will fail: The resulting system will allow to *block* messages on app_tx because

$$
\begin{array}{l}
(M3 \mid\mid\mid M1ACK) \parallel (ABP1\_TX \mid\mid\mid ABP1\_RC) \parallel (SEC1\_TX \mid\mid\mid SEC1\_RC) \\
\quad \setminus \{\mid sec\_tx, sec\_rc, sec\_ack\_tx, sec\_ack\_rc, abp\_tx, abp\_rc, abp\_ack\_tx, abp\_ack\_rc \mid\}
\end{array}
$$

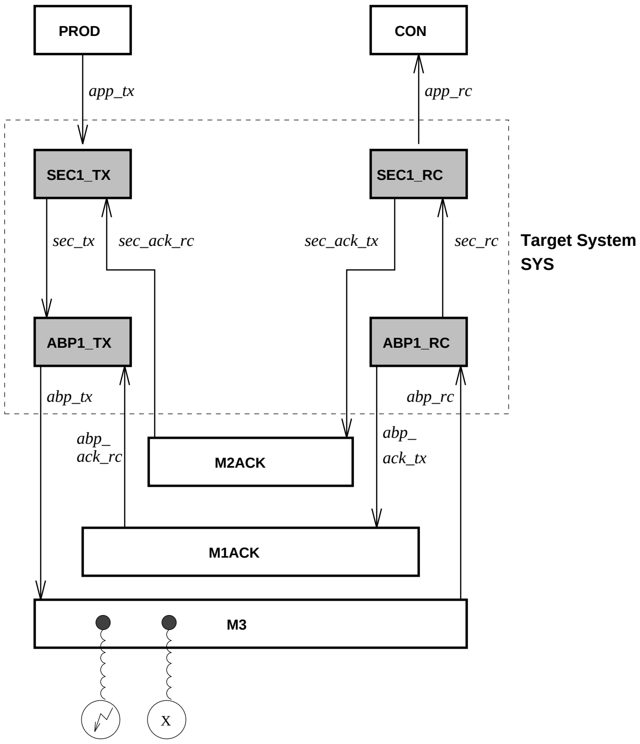

Figure 1.5: Naive combination of security layer and network layer with unreliable *and* insecure medium $M3$. (For the sake of simplicity, $M1ACK$ and $M2ACK$ are assumed to be ideal transmission media.)

diverges: There exist traces of unbounded length consisting only of events other than app_tx, app_rc. For example, consider the process $DivTrace$:

$$
\begin{aligned}
&\mathit{DivTrace} = \\
&\quad \mathit{app\_tx}?x \rightarrow \mathit{Div}0(x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Div}0(x) = \\
&\quad \mathit{sec\_tx}.\epsilon(x) \rightarrow \\
&\quad \mathit{abp\_tx}.(0, \epsilon(x)) \rightarrow \mathit{abp\_rc}.(0, \mathit{modify}(\epsilon(x))) \rightarrow \mathit{sec\_rc}.\mathit{modify}(\epsilon(x)) \rightarrow \\
&\quad \mathit{sec\_ack\_tx}.\mathit{Modified} \rightarrow \mathit{sec\_ack\_rc}.\mathit{Modified} \rightarrow \mathit{abp\_ack\_tx}.0 \rightarrow \\
&\quad \mathit{abp\_tx}.(0, \epsilon(x)) \rightarrow \mathit{abp\_rc}.(0, \epsilon(x)) \rightarrow \\
&\quad \mathit{abp\_ack\_rc}.0 \rightarrow \mathit{abp\_ack\_tx}.0 \rightarrow \mathit{abp\_ack\_rc}.0 \rightarrow \\
&\quad \mathit{sec\_tx}.\epsilon(x) \rightarrow \\
&\quad \mathit{abp\_tx}.(1, \epsilon(x)) \rightarrow \mathit{abp\_rc}.(1, \mathit{modify}(\epsilon(x))) \rightarrow \mathit{sec\_rc}.\mathit{modify}(\epsilon(x)) \rightarrow \\
&\quad \mathit{sec\_ack\_tx}.\mathit{Modified} \rightarrow \mathit{sec\_ack\_rc}.\mathit{Modified} \rightarrow \mathit{abp\_ack\_tx}.0 \rightarrow \\
&\quad \mathit{abp\_tx}.(1, \epsilon(x)) \rightarrow \mathit{abp\_rc}.(1, \epsilon(x)) \rightarrow \\
&\quad \mathit{abp\_ack\_rc}.1 \rightarrow \mathit{abp\_ack\_tx}.1 \rightarrow \mathit{abp\_ack\_rc}.1 \rightarrow \mathit{Div}0(x)
\end{aligned}
$$

$DivTrace$ produces a trace which is correct with respect to the "naive system". This can be proven using FDR by checking that $DivTrace$ is a correct trace refinement of

$$(M3 \mid\mid\mid M1ACK) \parallel (ABP1\_TX \mid\mid\mid ABP1\_RC) \parallel (SEC1\_TX \mid\mid\mid SEC1\_RC)$$

The problem arises as follows: The message x sent by $PROD$ via app_tx is modified during the first transmission on $M3$, identified by bit '0'. As a consequence the control message $sec\_ack\_tx.Modified$ is sent back to $SEC1\_TX$. Meanwhile a re-transmission is initiated by the ABP layer because the acknowledgement associated with the first message has not yet been received by ABP_TX. This message passes $M3$ without any modifications, but it is useless, because ABP_RC discards it on account of the same bit identification '0' as in the first transmission. Next ABP_TX receives an acknowledgement for the first transmission, because is has been carried out successfully, as far as the ABP layer is concerned. Now the re-transmission $sec\_tx.\epsilon(x)$ takes place, initiated by SEC_TX as a response to the $Modified$ control message. Again, this message is modified, this time carrying bit '1'. This procedure can be continued endlessly, and the hypothesis that at most $maxModifiedOrLost$ consecutive messages may be corrupted is never violated, if $maxModifiedOrLost > 0$.

Note that it does not help to swap the ABP layer and the security layer: Since $SEC1\_TX$ has been built under the assumption that messages will only be changed, SEC_TX might wait forever for an acknowledgement on channel sec_ack_rc after a message has been lost on $M3$. This will prevent the ABP layer from re-transmitting the message.
□

In the next chapter, based on the systematic approach for the development of dependable systems, a correct protocol for the combined fault hypothesis and security threat will be presented.

2. A Framework for the Development of Dependable Systems

2.1 Overview

This chapter describes a formal approach for the systematic development of dependable systems. The necessity for such an approach is illustrated in Section 2.2 by means of an industrial project in the field of railway interlocking systems, which is a typical example of control systems requiring the full scale of dependability properties. Motivated by the introductory example presented in Section 1.2 we claim that any formal method suitable for the development of dependable systems should be applied according to a *development standard* (Section 2.3)[1]. Such a standard provides a re-usable framework for the specification and verification activities to be performed during the various phases of system development. It does not require a specific method to be applied but describes the activities to be performed and objects to be produced during the development stages. Tailored for the specific problems related to the development of dependable systems, it ensures that design errors as the one discussed in 1.2 are very unlikely to occur. Indeed, today many researchers and practitioners are convinced that a formal method can only be successfully applied in an industrial context if supported by both tools and a development standard.

The work presented in this chapter summarises and partly extends the results described in [78], using CSP [44] as the underlying method for the specification and verification of distributed systems. To overcome the problems specifically related to dependability in a systematic way, we make use of a method developed by Schepers [104] in the context of fault-tolerant systems. We will show in Section 2.4 how his method may be extended to the field of dependability in general, transforming the design obligation to develop a dependable system into the design obligation for an "ordinary" system. As for the development standard, we will focus on the *Vorgehensmodell (V-Model)* [112] which is authorised by the Germany Ministry of the Interior and considered as the state-of-the-art software development standard in Germany. Application of a standard helps to benefit from a formal method in the most systematic way during the software life cycle. Conversely, the formal approach will increase the insight about how the standard should be applied in an optimal way and what the documents informally introduced by the standard should describe to achieve a useful and reliable description of the system to be developed.

To illustrate the application of our approach, the case study of Chapter 1 will be re-worked in Section 2.5 in a systematic way, this time leading to a correct solution allowing to re-use at least one of the protocols introduced before to defend the system against isolated threats.

[1]In this context *"standard"* means either an *(inter-)national* standard regulating development procedures or a *"customised" company* standard, which is usually an instantiation of an international or national standard for the specific type of tasks to be solved in the company.

2.2 Related Industrial Projects

The following example of a "real-world" project is presented to support the assertion that modern control systems should combine aspects of both "classical" dependability – i. e. safety, reliability, availability – *and* security. It has motivated the case study presented in Section 1.2, to be continued below. The example is typical for systems where classical dependability has always been an important issue, but the demand for security mechanisms is relatively new. Further examples are given in [78].

Distributed Railway Control System In Germany today, available electronic computer based railway control systems are centralised. One signal tower controls the state of all signals, points and level crossings of a specific area. Signals, points and level crossing barriers (or traffic lights) are directly wired to the signal tower. Each train is supervised by exactly one signal tower at a time. This technique is much too expensive for small private tramways, railway or underground networks. On the other hand, in the future these smaller networks will depend on highly automated control systems, because otherwise the operational costs would be too high. Therefore several European companies and research institutions investigate concepts for decentralised railway control.

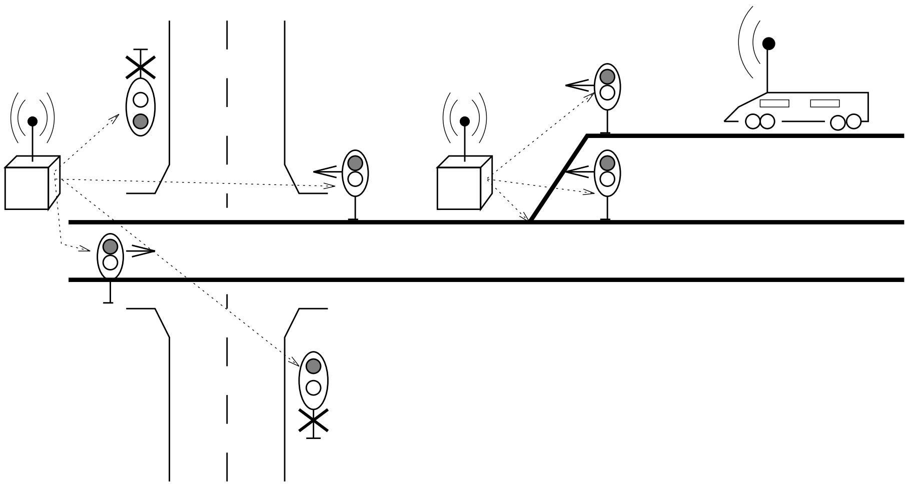

Figure 2.1: System overview of a distributed interlocking system.

In one of these concepts[2] the control logic is distributed on a collection of 'local' switch boxes and on the trains (Figure 2.1). Each switch box controls a small number of nearby signals, points and level crossings. It stores its local system state, supervises its local safety conditions and communicates with approaching trains. The control computer of a train will request local state information from (several) switch boxes and decide, based on the state information available, whether signals, points and barriers or traffic lights may be switched according to the request of the train or whether the train will have to wait at a signal. Data between trains and switch boxes is communicated via radio transmission, preferably making use of standard networks designed for the purpose of digital data transfer.

[2] The information in this paragraph is presented by courtesy of ELPRO LET GmbH. At present, I am working as a consultant for this company in the field of test automation and verification of components of interlocking systems. The concept is described in the proposal [26] in more detail.

This basic idea induces a variety of dependability requirements differing from those of centralised railway control system:

- The decision whether a train is allowed to pass a certain signal, point etc. is produced by a distributed algorithm which involves switch boxes "safe-guarding" their local safety requirements and the train control computer deducing the "global decision" from the local information of the switch boxes. The algorithm must never allow a transition into an unsafe state, even in the presence of component failures (safety requirement plus reliability requirement).
- Data exchange between trains and switch boxes has to apply protocols based on repreated package transmission to avoid the loss of data caused by accidental disturbance of the radio transmission (reliability requirement).
- The data transmission must also be protected against *systematic* manipulation by humans using radio transmission devices (security requirement).

Though the accidental physical disturbance of radio transmission will certainly be the threat with the highest probability, I believe that the idea of humans trying to manipulate the exchange of control signals does not represent a theoretical, but a very realistic threat. If hackers find it fascinating to plant viruses into other people's PCs (even though they cannot watch the effect in most cases), how much more fun would it be to stop, accelerate and re-direct a real train by "remote control", while watching the railway track! For these reasons, the distributed concept will introduce a new level of complexity. However, this does not represent an unmanageable problem, because the networks we consider here are much simpler structured and have a lower traffic rate than the big railways networks where the centralised concepts are used.

2.3 Standards for the Development of Dependable Systems

2.3.1 Overview

Software Development Standards provide frameworks for the activities to be performed and objects to be produced during the software life cycle. They describe the logical links between development objects (specification documents, software code, test data etc.) and specify the activities to be exercised on the objects (development, review, verification etc.), as well as their "synchronisation" during the development life cycle. Ideally, the application of such a framework will help to meet the quality requirements applicable for the target system in the most systematic way. Standards admit a variety of methods – informal, semi-formal or formal – to be applied during the phases of system development. Therefore they are of informal nature themselves, written in natural language. However, they usually suggest the characteristics that should be present in a method, when applied to develop a system of certain criticality and according to pre-defined quality requirements. The relationship between standards, methods and tools is depicted in Figure 2.2.

Examples for standards with relevance in the field of dependable systems are

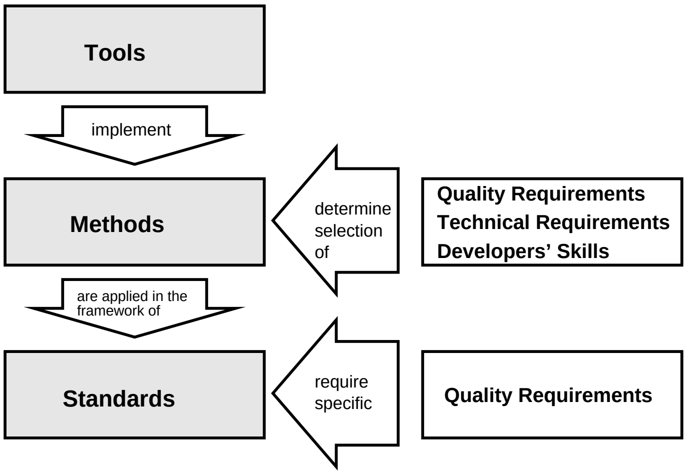

Figure 2.2: Relationship between standards, methods and tools.

- **ISO 9000-3** [24] At present this is the most well known and at the same time the most general international standard providing rules with respect to organisational aspects of software projects and standardisation and documentation of the complete software development life cycle. Apart from rules concerning the process of software development itself, it emphasizes the regulation of *every* business process which may have a direct or indirect impact on the quality of software products developed in a company.
- **ITSEC (Information Technology Security Evaluation Criteria)** [47] This standard focuses on the security aspect of dependable systems. It will be discussed in more detail in Chapter 5.
- **RTCA DO-178-B (Software Considerations in Airborne Systems and Equipment Certification)** [22] The international development standard for software in civil aircrafts. Apart from the general requirements regarding project management, configuration management and software development, its main issues are the selection and justification of validation, verification and test methods, safety aspects and reliability aspects. The CIDS system [1] discussed in [85] has been developed according to this standard.
- **V-Model** *('Vorgehensmodell')* [112] The standard for the development of software-based systems authorised by the German ministry of the interior.

Each of these standards is written in natural language. The objects and concepts used are informally defined and therefore subject to different interpretations. While each standard suggests the application of formal methods for the description and verification of the most critical system components, only few attempts have been made to show how the objects and concepts described in the standard should be related to the objects and concepts to be usually encountered when applying formal methods. With respect to the full scale of dependability issues, our approach to properly relate these different "worlds" appears to be a new enterprise. Similar investigations are currently carried out by Ravn and Stavridou [96, 97] with respect to the British Ministry of Defence Interim Standard MoD 00-55 [65]. In this chapter, we will discuss the regulations for software development according to the *V-Model*, because it may presently be regarded as one of the most advanced standards in Europe. Nevertheless, several adjustments of the standard will be made concerning the structure and the contents of the documents introduced by the *V-Model*. These changes will be introduced in Section 2.3.2 and justified while introducing the formal approach in Section 2.4.

In Chapter 5 a similar approach will be described, this time focusing on security aspects only and referring to the ITSEC.

2.3.2 System Development According to the V-Model

Process Model

Similar to [24, 22], the *V-Model* adopts the well-accepted concept to model the whole system development procedure as a system of cooperating "meta" processes: Each team performs a well-defined task. In order to complete the task, inputs (documents, code, test data etc.) are received from the environment (e. g., the customer) and from other teams. After having processed the inputs, the corresponding outputs are distributed to the teams, to the customer and other external groups. The standard project structure comprises four cooperating processes: *Software Development Team, Project Management Team, Quality Assurance Team* and *Configuration Management Team*. In the following we will focus on the tasks to be performed by the software development and the quality assurance teams in the context of the development of dependable systems.

Levels of Abstraction

To make the development of large and complex systems more feasible, they are described by means of a collection of documents describing different levels of abstraction and associating a modular structure with each level. In the *V-Model* five levels of modularisation and abstraction are proposed for the full description of a system. They are informally introduced as *(1) System Level:* The top-level description of the whole target system without distinctions between software and hardware. Data and functions are described on the end user's level of abstraction. Interfaces to existing components in the environment (e. g., another computer) are described on the level of abstraction which is "seen" by the software to be developed for the target system. For example, if only application software has to be developed in a system layered according to the OSI model, the interface to an existing network is described on the level of abstraction offered by the send- and receive services provided by the OSI session layer for the application. *(2) Sub-System Level:* Large systems (e. g., a wide area network) may be decomposed into sub-systems (e. g., a local area network in the wide area network), in general without changing the level of abstraction for the representation of data, functions and interfaces. *(3) Segment Level:* This is a further modularisation step which partitions (sub-)systems into well-defined components (e. g. one computer in a network). On this level the segments consisting of hardware only are separated from those to be built out of both hardware and software. *(4) SW Configuration Item Level:* The software to be allocated in a segment is decomposed into different configuration items, each item representing a software sub-system (e. g., the processes associated with a specific layer of the OSI model which will be allocated in the segment). This is usually the starting point from where to change also the level of abstraction for the representation of data and associated functions. *(5) Component Level:* A part of a configuration unit, performing a well-defined service (e. g., one task of the configuration unit). Several component layers may be "inserted" to refine a top-level component (e. g., tasks might be decomposed into threads). *(6) Module Level/Data Level:* This is the lowest level of abstraction and modularisation to be distinguished according to the *V-Model* (e. g., a sequential function or a database table with associated attribute definitions).

Documents Associated With Each Level

The *V-Model* describes the documents to be associated with each level of abstraction and the topics to be covered by each document. In the context of this chapter two types of documents are of interest which appear (with slightly different naming and varying contents) on every level: *(1) Requirements Description:* This document type describes the functions, data and dynamic behaviour of an object associated with a specific level. In addition, boundary conditions restricting the class of possible realisations for the object are documented. *(2) Architecture Description:* This is a design structure which decomposes the object under consideration and/or refines its data structures. The process of decomposition introduces new objects to be associated with a lower level, as well as interfaces between them. Each new object is associated with its own lower-level requirements description. In this way, the alternation between requirements and architecture documents can be recursively applied to the decomposition tree from system to module level.

To illustrate the contents of requirements and architecture documents, let us look at the system level.

System Requirements The system requirements document should describe *what* the system is supposed to do without telling us *how* this can be achieved. Specifically, the document describes the following aspects: *(1) Operational Environment:* The external interfaces and hypotheses about the normal behaviour of the environment. *(2) Application Requirements:* A specification of the functions, data and dynamic behaviour of the complete application, as seen by the end user. The application requirements describe this behaviour in relation to the operational environment (1). This means that other types of behaviour have to be expected, if the characteristics of the operational environment are changed. *(3) Threat Analysis (External):* A specification of those possible deviations from normal behaviour which are to be expected *in the environment* (e. g., crash of external computers, security attacks by unauthorised users from outside the system, corrupted data at an input interface). For each threat the list of dependability requirements it might impair is documented. Note that since at this stage nothing is known about the *internal* structure of the system, only *external* threats can be specified. *(4) Dependability Requirements:* A list associating items from the application requirements with safety, reliability/availability and security requirements. This describes the degree of protection to be established for each of these items in case of a deviation of the environment from normal behaviour. *(5) Risk Analysis (External):* An estimator associated with each threat, specifying the probability for the corresponding exceptional behaviour to occur.

This is not the complete list of items to be covered by the system requirements document, but the others described in [112] are not relevant in our context. It is more important to note that we have deviated from the description [112] in the following ways:

- According to the *V-Model*, security requirements and reliability requirements are described in different parts of the document, while safety requirements are subsumed under reliability and availability is not mentioned at all. I think that this is inappropriate, since it will be illustrated below that all dependability aspects can be covered by a unified approach. Therefore I have decided to introduce the general term *Dependability Requirements* where safety, reliability, availability and security issues should be described.

- The *V-Model* does not distinguish between external and internal threat/risk analysis. I regard this as an important aspect, because in general designers will have no influence on external threats, while they can influence the internal ones by means of their design decisions.

System Architecture The main topics to be covered by the system architecture document are *(1) System Structure:* The system is decomposed into sub-systems, segments or software configuration items. *(2) External Interfaces:* The external interfaces described in the system requirements document are allocated to the corresponding components of the system structure. *(3) Internal Interfaces:* The internal communication paths between the new components are defined. This gives rise to interface specifications for the requirements documents associated with each component using the interface. *(4) Requirements Allocation:* Each application requirement is associated with one or more components of the system structure which have been designed to implement the requirement. This induces application requirements specifications for each component. *(5) Dependability Requirements Allocation:* The dependability requirements are mapped to corresponding components designed in the system structure. This induces dependability requirements for each of these components. *(6) Threat Analysis (Internal):* The decomposition introduces new interfaces and new components, therefore new threats may arise in addition to the external threats (e. g., component failure, security leak inside the system). *(7) Risk Analysis (Internal):* associated with the internal threat analysis. *(8) Dependability Concept:* It is explained how the system dependability requirements can be achieved by means of the chosen system structure in combination with the requirements defined for each component.

Here we have altered the suggestions of the *V-Model* in the following way:

- In [112] only the description of the security concept is required. Consistent with the change proposed for the system requirements, we will describe the complete dependability concept instead.
- The internal threat/risk analysis is missing in the *V-Model*. This also holds for the requirements and architecture documents associated with the lower levels of description. This seems to be an important aspect overlooked by the authors of [112]: Each design decision may introduce new threats to the dependability requirements. Therefore it is necessary to perform a new threat analysis at each level of decomposition.

2.4 A Formal Approach for the Development of Dependable Systems

2.4.1 Selection Criteria for the Formal Method

When choosing a suitable formal method for the development of dependable systems of the type introduced in the case study in Chapter 1, the following selection criteria have been considered:

- **Specification of distributed systems:** Since the problem belongs to the field of distributed systems, the underlying method should allow to specify aspects of concurrency, synchronisation etc. in a suitable way.
- **Compositional and modular refinement:** The technique of describing the target system by means of stepwise decomposition, as suggested by the *V-model*, is related to refinement in a natural way. Moreover, there are two types of refinement that typically occur during the development process. They have been explored in detail by Zwiers [120]: If the designer has full freedom in the decomposition of a system component, being allowed to choose both the architecture and the specifications of the lower-level components created in the decomposition step, we speak of *compositional refinement*. If, on the other hand, existing components (e. g., off-the-shelf software) have to be used in the decomposition step we are confronted with pre-defined specifications and have to adjust the architecture and the specifications of new modules in such a way that the pre-defined ones can be integrated. This situation is called *modular refinement*. Since most software projects require both the development of new components and the integration of existing ones, we require that a suitable method should allow both compositional and modular refinement.
- **Dependability features:** The formal method should allow to derive systematically the description of system behaviour in presence of (combinations of) threats.

These criteria led to the selection of CSP [44], enriched by Schepers' theory [104] for the characterisation of fault hypotheses and their impact on distributed systems: The CSP process algebra has been explicitly designed to describe and verify distributed systems, several of its semantics definitions are compositional and the associated proof theories support both modular and compositional refinement. Schepers' approach introduces one of the few available techniques allowing to model *combinations* of threats and their resulting impact on system behaviour in a formal way. An additional motivation for this selection was that the method and the tools for test automation in the field of safety-critical reactive systems described in Chapter 4 are also based on CSP. As a consequence, the formal specification and verification approach described here may be consistently used together with the test and validation techniques of Chapter 4.

Various denotational and operational semantics have been defined for CSP. The formalisations and results described below hold

1. in the *trace model* of CSP, as described in [44],
2. in the *failures-divergence model* of CSP, as described in [44],

3. in the *timed CSP model* in the extent as introduced in [104].

The first model is useful to describe safety aspects only. The second model may be used to specify and verify both safety and lifeness properties in the untimed case, and the last one should be applied when real-time aspects have to be explicitly considered as well.

In the sections to follow, we will associate formal notions with the system requirements and the system architecture, as defined by the *V-Model*. Moreover, it will be indicated how the formalisation on system level may be continued in a recursive way to cover the requirements and architecture descriptions of lower levels as well. It might be helpful to read in parallel the case study in Section 2.5, because it will illustrate most of the formalisations introduced below.

2.4.2 System Requirements

In this section formalisations for the following items of the system requirements document will be presented:

- operational environment
- application requirements
- threat analysis (external)
- dependability requirements

The risk analysis is not covered in our approach, because it requires other (e. g., statistical) methods outside the scope of the formal methods applied here.

Operational Environment

To formalise the notion of an operational environment recall that a *context* is a CSP term $\mathcal{C}(X)$ with one free identifier X. Since the CSP operators $\rightarrow$, $\|$, $|||$, $\Box$, $\sqcap$, $;$, $\backslash$ used in a process term are continuous, every context can be regarded as a continuous mapping from CSP processes X to CSP processes $\mathcal{C}(X)$ (see [71, p.160]). In order to formalise the operational environment, we use a special context $\mathcal{E}(SYS)$ with the following interpretation:

1. The free identifier SYS denotes the *target system* to be "plugged into" the environment $\mathcal{E}$.
2. Every other process identifier E_i appearing in the term $\mathcal{E}(SYS)$ is an *environment process* which is considered relevant with respect to the operation of SYS, but is not a part of SYS itself.
3. The CSP operators used to define the term $\mathcal{E}(SYS)$ describe the *architecture* of the environment.

Each process parameter E_i in the term $\mathcal{E}(SYS)$ is associated with

- its alphabet αE_i, as far as visible to the outside world,
- a specification E_i **sat** $S_{E_i}(h)$ representing the *normal behaviour* of E_i when the trace model of CSP is appropriate for specification and verification purposes,

- a specification E_i **sat** $S_{E_i}(h, R)$ representing the *normal behaviour* of E_i when the failures(-divergence) model of CSP is appropriate for specification and verification purposes.

We will use this behavioural specification style also in the other specifications to follow. In P **sat** $S(h, R)$, $S(h, R)$ is a predicate with free variables h and R. h denotes a trace which can be executed by P, and R is a refusal set of P/h, the process state of P after having executed trace h. This holds for behavioural specifications in the failures-divergence and in the timed CSP model. If only properties about traces are described, specifications $S(h)$ are used, possessing only the trace parameter as free variable. As a consistency condition it is required that

$$ch(S(h, R)) \subseteq ch(P)$$

i. e., every channel referenced in the predicate $S(h, R)$ is also a channel of the process P.

To complete the specification of the operational environment, we will introduce a set

$$I \subseteq \bigcup_i \alpha(E_i)$$

denoting the *interface* between the environment and the target system SYS.

It is important to note that for the relevant parts of the environment, the system requirements specify behaviour *plus* architecture. Otherwise a threat analysis would become impossible, as shall be explained below.

Application Requirements

Our objective is to develop a system SYS subject to a behavioural specification $S_{SYS}(h, R)$ *when operating in the specified environment* $\mathcal{E}$. Therefore the application requirements are formalised as

$$\mathcal{E}(SYS) \ \mathbf{sat} \ S_{SYS}(h, R)$$

which is again interpreted as the *normal* system behaviour. The consistency conditions

$$I \subseteq \alpha(SYS)$$

$$ch(S_{SYS}(h, R)) \subseteq ch(SYS)$$

require that SYS must contain the pre-defined interface and its specification may only refer to SYS-events. If $ch(S_{SYS}(h, R)) \subseteq I$ the application requirements are called a *black box* specification, since they describe only the system behaviour visible at the interface.

Threat Analysis (External)

The objective of the external threat analysis is to specify the possible deviations from normal behaviour of the environment. In the CSP world, the only observable objects are traces and – in the failures-divergence model or for timed CSP – refusals. Therefore a threat can only refer to possible deviations from the specified characteristics of traces and refusals. To this end Schepers [104] has introduced assertions of the type

$$P \textbf{ sat } \Delta(h, h', R, R')$$

where Δ is a predicate with free variables h, h', R, R', to be called a *threat*[3]. In this notation (h, R) are interpreted as the trace/refusal pairs representing normal behaviour, whereas (h', R') denote the trace and refusal of the exceptional behaviour. Informally speaking, the threat $\Delta(h, h', R, R')$ is a predicate relating the exceptional behaviour to the normal behaviour. The following consistency conditions are required:

- Threats are reflexive relations with respect to (h, R): $\models \Delta(h, h, R, R)$
- Threats are prefix-closed with respect to the trace h':

  $$\models (\Delta(h, h', \emptyset, \emptyset) \wedge s' \leq h') \Rightarrow (\exists\, s \bullet s \leq h \wedge \Delta(s, s', \emptyset, \emptyset))$$

- Threats are subset-closed with respect to refusals R':

  $$\models (\Delta(h, h', R, R') \wedge U' \subseteq R') \Rightarrow (\exists\, U \bullet U \subseteq R \wedge \Delta(h, h', U, U'))$$

- The channels referenced in $\Delta(h, h', R, R')$ have to be channels of the process threatened by $\Delta(h, h', R, R')$.

Since – as demonstrated in the example of Chapter 1 – such a predicate can become very complex as soon as more than one threat is involved, it is useful to determine $\Delta(h, h', R, R')$ in a systematic stepwise procedure:

Threat Analysis – Step 1: Definition of Isolated Threats For each environment process E defined by the context $\mathcal{E}$ and each isolated threat having impact on the process, set up a predicate specifying the acceptable behaviour of E *in presence of this isolated threat.* This results is a list of predicates

$$\Delta_{E,j}(h, h', R, R')$$

ranging over environment processes E and isolated threats indexed by j.

Threat Analysis – Step 2: Compositional Ordering of Threats In preparation of the next step, where the resulting overall threat shall be derived from the isolated threats, it is necessary to ensure that a certain *Compositional Ordering* of the threats can be obtained. A systematic derivation of the resulting threat will only be possible if the environment architecture suggests that the isolated threats $\Delta_{E,i}$ for an environment process E can be arranged in an order $\langle \Delta_{E,j_1}, \Delta_{E,j_2}, \Delta_{E,j_3}, \ldots \rangle$ such that each Δ_{E,j_ℓ} can be assumed to act on *E with the threats $\langle \Delta_{E,j_1}, \ldots, \Delta_{E,j_{\ell-1}} \rangle$ already present.* There seems to be no way to formalise the condition for such an ordering to exist, because no formal reference document is available

[3] In [104] the term *failure hypothesis* is used for $\Delta(h, h', R, R')$, since Schepers uses these predicates only for the description of exceptional behaviour in the context of fault tolerance. Failure hypotheses are typically denoted by $\chi(h, h', R, R')$ in [104].

on the system level from where such an ordering could be verified. As a consequence, it depends completely on the insight of the analysts how to find such an ordering and how to justify it in an intuitive way. If no compositional ordering of threats may be justified, the overall threat has to be derived from scratch without making use of isolated threat specifications.

Threat Analysis – Step 3: Derivation of the Resulting Threat If a compositional ordering $\langle \Delta_{E,j_1}, \Delta_{E,j_2}, \Delta_{E,j_3}, \ldots \rangle$ of threats can be found in step 2, the resulting threat may be derived by the *composition of threats* Δ_1, Δ_2 using the *threat introduction operator* $\wr$ defined by Schepers [104] as

$$(\Delta_1 \wr \Delta_2)(h, h', R, R') \equiv_{df} (\exists\, s, U \bullet \Delta_1(h, s, R, U) \wedge \Delta_2(s, h', U, R'))$$

$(\Delta_1 \wr \Delta_2)$ expresses the situation where the threat Δ_2 acts "on top of" threat Δ_1: Δ_1 refers to h representing normal behaviour and admits a deviation s. Then Δ_2 specifies exceptional behaviour h' *in reference to* s, which represents the exceptional behaviour caused by Δ_1.

Threat Analysis – Step 4: Derivation of Specifications in Presence of Threats After completion of Step 3, each process E of the operational environment is associated with

- a specification of *normal behaviour* E **sat** $S_E(h, R)$, as obtained in the operational environment specification,
- an *overall threat* $\Delta_E(h, h', R, R')$ associated with E and derived according to steps 1 to 3 of the threat analysis[4].

The last step of the threat analysis is devoted to the derivation of the specification of each environment process *in presence of its overall threat*. To this end, we can use the *threat introduction rule*[5] given by Schepers [104, p. 84] for his proof theory for *failure prone processes*:

$$\frac{E \ \textbf{sat}\ S_E(h, R)}{(E \wr \Delta_E)\ \textbf{sat}\ (S_E \wr \Delta_E)(h', R')}$$

In this rule, $(E \wr \Delta_E)$ denotes the process capable of both normal behaviour as performed by E and exceptional behaviour as defined by Δ_E. In analogy to the composition of threats, the *composition of specifications* $S(h, R)$ *and threats* $\Delta(h, h', R, R')$ is defined by

$$(S \wr \Delta)(h', R') \equiv_{df} (\exists\, h, R \bullet S(h, R) \wedge \Delta(h, h', R, R'))$$

Note that since (h, R) are bound by the existential quantifier, this is again an "ordinary" specification with only one pair (h', R') of free variables.

Example 2.1 To provide an example where the overall threat cannot be represented by a compositional ordering of the isolated ones, let us assume that the ideal system in absence of threats should do nothing but execute the trace

$$h = \langle c.1, c.1, c.1, \ldots \rangle$$

[4]If E is not subject to exceptional behaviour, the threat can be assumed to be $\Delta_E(h, h', R, R') \equiv h = h' \wedge R = R'$.

[5]This rule is called *failure hypothesis introduction* in [104].

Assume further that the implemented system is threatened by[6]

$$
\begin{aligned}
\Delta_1(h,h') \equiv_{df}\ & ch^*(h') = ch^*(h) \\
& \wedge (\forall\, i : 1\,..\,\#h' \bullet h'(i) = h(i) \vee val(h'(i)) = val(h(i)) + 2) \\
\Delta_2(h,h') \equiv_{df}\ & ch^*(h') = ch^*(h) \\
& \wedge (\forall\, i : 1\,..\,\#h' \bullet h'(i) = h(i) \vee val(h'(i)) = 4 * val(h(i)))
\end{aligned}
$$

but the combined threats interact in such a way that they mutually exclude each other in each step, so that the overall threat is described as

$$
\begin{aligned}
\Delta_0(h,h') \equiv_{df}\ & ch^*(h') = ch^*(h) \wedge (\forall\, i : 1\,..\,\#h' \bullet h'(i) = h(i) \vee \\
& val(h'(i)) = val(h(i)) + 2 \vee val(h'(i)) = 4 * val(h(i))) \\
\equiv\ & (\forall\, i : 1\,..\,\#h' \bullet h'(i) \in \{c.1, c.3, c.4\})
\end{aligned}
$$

Then, applying the definition of the threat introduction operator $\wr$, we get

$$
\begin{aligned}
(\Delta_1 \wr \Delta_2)(h,h') &\equiv (\forall\, i : 1\,..\,\#h' \bullet h'(i) \in \{c.1, c.3, c.4, c.12\}) \\
(\Delta_2 \wr \Delta_1)(h,h') &\equiv (\forall\, i : 1\,..\,\#h' \bullet h'(i) \in \{c.1, c.3, c.4, c.6\})
\end{aligned}
$$

and thus neither $(\Delta_1 \wr \Delta_2)(h,h')$ nor $(\Delta_2 \wr \Delta_1)(h,h')$ are equivalent to $\Delta_0(h,h')$
$\square$

Dependability Requirements

The informal definition of dependability requirements given by the *V-Model* may be rephrased as a specification describing the *maximum deviation from normal behaviour* which may be tolerated for certain objects of the application. As pointed out above, these objects may only be traces and refusals in the CSP context, and we may use the same formalism as in the threat analysis above: A dependability requirement is a threat predicate

$$\Delta_{SYS}(h, h', R, R')$$

satisfying

$$ch(\Delta_{SYS}(h, h', R, R')) \subseteq ch(SYS)$$

and interpreted as

> *Even in presence of exceptional environment behaviour or exceptional internal behaviour, the specification $(S_{SYS} \wr \Delta_{SYS})(h', R')$ has still to be observed.*

Summary of the System Requirements

Before tackling the system architecture document, let's summarise what has been achieved during the development of system requirements. At first, the following activities were performed:

1. definition of a *context* $\mathcal{E}(X)$ describing the architecture of the environment, as far as relevant for the development of the target system,

[6] For any event $c.x$ with channel c and value x, $ch(c.x) = c$ and $val(c.x) = x$ denote the projections on the channel and the value, respectively.

2. specifications E **sat** $S_E(h, R)$ of the *normal behaviour of environment processes* E defined by the context $\mathcal{E}(X)$,
3. specification $\mathcal{E}(SYS)$ **sat** $S_{SYS}(h, R)$ of the *normal system behaviour*,
4. specifications of (lists of) *isolated threats* $\Delta_{E,j}(h, h', R, R')$ for the environment processes E introduced by $\mathcal{E}(X)$,
5. specification of *dependability requirements* $\Delta_{SYS}(h, h', R, R')$ describing the acceptable deviations of the target system from normal behaviour, as far as visible at the system interface.

In this list the items 1 to 3 will be present in *any* system development procedure. Only items 4 and 5 are specific for the development of *dependable* systems. The inputs 1 to 5 were transformed into

1. specifications $(E \wr \Delta_E)$ **sat** $(S_E \wr \Delta_E)(h', R')$ of the *acceptable behaviour of environment processes* E defined by the context $\mathcal{E}(X)$
2. a modified context $(\mathcal{E} \wr \Delta)(X) = \mathcal{E}(X)[E/(E \wr \Delta_E), \ldots]$, where every process parameter E used in $\mathcal{E}(X)$ has been replaced by the process $(E \wr \Delta_E)$ showing acceptable behaviour in presence of threats
3. a predicate $(S_{SYS} \wr \Delta_{SYS})(h', R')$ defining the *acceptable system behaviour* in presence of all possible threats. This specification summarises the normal behaviour specification plus the dependability requirements

Analysis of $(E \wr \Delta_E)$, $(\mathcal{E} \wr \Delta)(X)$ and $(S_{SYS} \wr \Delta_{SYS})(h', R')$ shows:

> *The obligations associated with the system requirements for dependable systems can be transformed into "ordinary" design obligations, containing the results of the threat analysis and the dependability requirements as integral parts of the behavioural specifications for environment processes and target system.*

Note that in our approach the threats have *not* been associated with the corresponding dependability requirements, as suggested in the *V-Model*: At the present stage of the development, the relation between the objects to be protected and the objects threatened by the environment is not clear because the architecture document is still missing. Therefore my suggestion is to describe the relation between threats and dependability requirements in the architecture document.

2.4.3 System Architecture

We will now describe the formal treatment of the items to be covered by the system architecture document, as has been informally defined by the *V-Model*. We claim that only the items

- System Structure
- Interfaces
- Internal Threat Analysis

- Internal Risk Analysis

have any formal relevance for the development and verification process. The *Correspondence between Application Requirements and Architecture* and the *Dependability Concept* only serve to illustrate the system concept, as will be made clear in the following paragraphs. As in the description of the system requirements document I will skip the *Internal Risk Analysis*.

System Structure

The target system SYS is decomposed into an *implementable* collection of cooperating processes. Formally speaking, the system structure is a CSP equation

$$SYS = \mathcal{A}_{SYS}(P_1, \dots, P_n)$$

where $\mathcal{A}_{SYS}$ is a continuous function mapping processes $(P_1, \dots, P_n)$ to a new CSP process by making use of the CSP operators. The processes P_i are components of a lower level, e. g., sub-systems. However, the terms *sub-system, segment, SW-configuration item* etc. do not have any formal meaning; their objective is only to aid the intuition of the reader trying to understand the specification document.

Interfaces

In general, the definition of the system structure will also introduce new channels, whose channel alphabets constitute the internal interface descriptions. Internal plus external interfaces are completely defined by the alphabets $\alpha(P_i \setminus L_{P_i})$, denoting the events visible at the interface of each process, while L_{P_i} denotes the set of internal P_i-events. Since one of the objectives of the architecture document is to completely establish the interface between target system and environment, we demand that the interface set I should be contained in the component alphabets, i. e.,

$$I \subseteq \bigcup_i \alpha(P_i)$$

Component Specifications

Each lower-level component P_i (sub-system, segment etc.) has to be equipped with a specification of its *normal* behaviour,

$$P_i \textbf{ sat } S_{P_i}(h, R)$$

This specification will be located in the component requirements document associated with P_i. However, it has to be developed during the definition of the system architecture, because otherwise the architecture cannot be verified against the system requirements.

Threat Analysis (Internal)

As is well-known in the context of dependable systems, a general refinement step may introduce new *internal threats* that could not be captured on system requirements level, *because they completely depend on the architecture selected in the refinement step.* As a consequence a new threat analysis and associated risk analysis have to be performed after the definition of the system structure. Though specific boundary conditions ensuring that dependability will be preserved under refinement are of great interest, especially in applications to security (see [101]), we think that in general it has to be accepted that the selection of a specific architecture may introduce new threats and lead to a "feed-back loop" between threat analysis and revision of refinement decisions.

The distinction between external and internal threat analysis is justified by the fact that the external threats are caused by the environment processes whose acceptable behaviour cannot be influenced by the system designer, whereas internal threats may be altered by means of architectural decisions.

Fortunately, the internal threat analysis may be treated in analogy to the external threat analysis described above: Starting with an association of isolated threats $\Delta_{P_i,j}$ for each process P_i, the threat analysis will derive specifications

$$(P_i \wr \Delta_{P_i}) \textbf{ sat } (S_{P_i} \wr \Delta_{P_i})(h, R)$$

which again constitute ordinary specifications of the acceptable behaviour of each P_i.

Dependability Concept

All information needed to verify the correctness of the architecture decisions is contained in the definition of $\mathcal{A}_{SYS}$ and the specifications of the lower-level components P_i including internal threats. As a consequence, the dependability concept will not add anything to the architecture document which is essential for specification or verification purposes. It is good design practice, however, to "separate concerns" in specifications. If a global specification of $\mathcal{C}((P_1 \wr \Delta_{P_1}), \ldots, (P_n \wr \Delta_{P_n}))$ can be separated into a conjunction of predicates $App(h, R)$ and $Dep(h, R)$ and the assertions expressed by $Dep(h, R)$ can be shown to be responsible for the preservation of dependability requirements while $App(h, R)$ expresses properties of the application system, then $Dep(h, R)$ will be called the *(formal) dependability policy* of the system.

Requirements Allocation

This is a means to *trace* the system requirements in the system architecture. In practice it is achieved by maintaining tables listing requirements features and associated components in the architecture which have been intended to implement the requirement. This is only an informal aid, since our formal approach offers the opportunity to *prove* that the architectural decisions have been correct, as can be seen in the next section. In the context of formal methods, a requirements allocation document would be a table listing the premises for the proofs which are necessary to demonstrate that the system decomposition designed really implies the requirements specification.

2.4.4 Verification on System Level

We wish to prove that the system structure selected is appropriate for the global development task in presence of external and internal threats. To this end, two things have to be shown:

1. If the lower-level processes perform according to their *normal* behaviour specification, their cooperation according to $\mathcal{A}_{SYS}$ results in a behaviour consistent with the *normal* behaviour specification *on system level*.

2. If the lower-level processes perform according to their *acceptable* behaviour specification, their cooperation according to $\mathcal{A}_{SYS}$ results in a behaviour consistent with the *acceptable* behaviour specification *on system level*.

At first glance, condition 1 might seem superfluous, because the formal definition of threats Δ requires that normal behaviour always satisfies the threat predicate, that is, $(h, R) \in Fail(P) \models \Delta(h, h, R, R)$. However, this requirement does not guarantee that the normal behaviour will really be *implemented* in the system: An implementation only showing *exceptional* behaviour would satisfy requirement 2, but certainly not be consistent with the customer's expectations.

Formally speaking, these considerations result in the following proof obligations:

1. System Verification Obligation – Normal Behaviour:

$$\frac{\begin{array}{l}(\forall\, i \bullet P_i \ \mathbf{sat}\ S_{P_i}(h, R))\\ (\forall\, j \bullet E_j \ \mathbf{sat}\ S_{E_j}(h, R))\end{array}}{\mathcal{E}(\mathcal{A}_{SYS}(P_1, \ldots, P_n)) \ \mathbf{sat}\ S(h, R)}$$

2. System Verification Obligation – Acceptable Behaviour:

$$\frac{\begin{array}{l}(\forall\, i \bullet (P_i \wr \Delta_{P_i}) \ \mathbf{sat}\ (S_{P_i} \wr \Delta_{P_i})(h, R))\\ (\forall\, j \bullet (E_j \wr \Delta_{E_j}) \ \mathbf{sat}\ (S_{E_j} \wr \Delta_{E_j})(h, R))\end{array}}{(\mathcal{E} \wr \Delta)(\mathcal{A}_{SYS}((P_1 \wr \Delta_{P_1}), \ldots, (P_n \wr \Delta_{P_n}))) \ \mathbf{sat}\ (S \wr \Delta_{SYS})(h, R)}$$

In proof obligation 2, $(\mathcal{E} \wr \Delta)(X) = \mathcal{E}(X)[E_1/(E_1 \wr \Delta_{E_1}), \ldots]$, where $[E_1/(E_1 \wr \Delta_{E_1}), \ldots]$ denotes the substitution of every environment process E_i showing only normal behaviour by its associated process $(E_i \wr \Delta_{E_i})$ showing acceptable behaviour.

Observe that on system level, we are confronted with the obligation to verify a *mixed modular and compositional refinement step*. The environment processes which cannot be altered according to the designer's preferences introduce modular refinement obligations. Compositional refinement, on the other hand, applies for the processes P_i created during the architecture design for the target system.

Furthermore, note that also the system verification obligation for acceptable behaviour is an ordinary proof obligation, where all references to dependability issues have been transformed into ordinary requirements during the threat analysis. Therefore the rules for the standard compositional proof theory of CSP can be applied to verify this assertion. Observe, however, that the creative process of finding suitable components P_i crucially depends on the existence of internal threats: If internal threats do not have to be taken into account, the system verification obligation for acceptable behaviour represents a standard proof obligation according to the *invent and verify paradigm*:

Given $(E_j \wr \Delta_{E_j})$ **sat** $(S_{E_j} \wr \Delta_{E_j})(h', R')$, $(\mathcal{E} \wr \Delta)(X)$ *and* $(S \wr \Delta_{SYS})(h, R)$ *as defined in the system requirements document, invent an architecture* $\mathcal{A}_{SYS}$ *and component specifications* P_i **sat** $S_{P_i}(h, R)$, *such that*

$$(\mathcal{E} \wr \Delta)(\mathcal{A}_{SYS}(P_1, \ldots, P_n)) \textbf{ sat } (S \wr \Delta_{SYS})(h, R)$$

holds.

The freedom to invent $\mathcal{A}_{SYS}$ and the specifications of P_i is severely restricted, if internal threats are present: Now the *invent and verify paradigm* has to be re-phrased as:

Given $(E_j \wr \Delta_{E_j})$ **sat** $(S_{E_j} \wr \Delta_{E_j})(h', R')$, $(\mathcal{E} \wr \Delta)(X)$ *and* $(S \wr \Delta_{SYS})(h, R)$ *as defined in the system requirements document, invent an architecture* $\mathcal{A}_{SYS}$ *and component specifications* P_i **sat** $S_{P_i}(h, R)$, *such that, taking into account the threats* Δ_{P_i} *induced by* $\mathcal{A}_{SYS}$,

$$(\mathcal{E} \wr \Delta)(\mathcal{A}_{SYS}((P_1 \wr \Delta_{P_1}), \ldots, (P_n \wr \Delta_{P_n}))) \textbf{ sat } (S \wr \Delta_{SYS})(h, R)$$

holds.

2.4.5 Recursive Application of the Development Procedure

After having established the system architecture with its corresponding component specifications, the approach can be recursively applied in top-down fashion:

1. For a given sub-system or lower-level component P and an associated specification P **sat** $S_P(h, R)$ for normal behaviour and $(P \wr \Delta)$ **sat** $(S_P \wr \Delta)(h, R)$ for acceptable behaviour develop a component architecture $\mathcal{A}_P(P_1, \ldots, P_k)$.
2. Associate specifications of normal behaviour for each P_i: P_i **sat** $S_{P_i}(h, R)$.
3. Analysing the architecture chosen for P, perform an internal threat analysis and derive the resulting overall threat Δ_{P_i} for each P_i.
4. Using the threat introduction rule, derive the specification of acceptable behaviours $(P_i \wr \Delta_{P_i})$ **sat** $(S_{P_i} \wr \Delta_{P_i})(h, R)$.
5. Prove the

   1. Component Verification Obligation – Normal Behaviour:

   $$\frac{(\forall\, i \bullet P_i \textbf{ sat } S_{P_i}(h, R))}{\mathcal{A}_P(P_1, \ldots, P_k) \textbf{ sat } S_P(h, R)}$$

   otherwise improve the architecture chosen for P or revise the requirements for the sub-components, P_i **sat** $S_{P_i}(h, R)$.
6. Prove the

   2. Component Verification Obligation – Acceptable Behaviour:

   $$\frac{(\forall\, i \bullet (P_i \wr \Delta_{P_i}) \textbf{ sat } (S_{P_i} \wr \Delta_{P_i})(h, R))}{\mathcal{A}_P((P_1 \wr \Delta_{P_1}), \ldots, (P_k \wr \Delta_{P_k})) \textbf{ sat } (S_P \wr \Delta)(h, R)}$$

otherwise improve the architecture chosen for P or revise the requirements for the sub-components, P_i **sat** $S_{P_i}(h, R)$.

7. Continue this process for each P_i until a sufficient level of decomposition has been reached.

In this stepwise refinement procedure, steps 3 and 4 are needed to transform the development obligation involving dependability issues into an “ordinary” development obligation. They are only necessary if the refinement step forces the designer to consider new dependability threats.

Note that from sub-system level downward only compositional refinement is needed, as long as the designer does not have to integrate re-usable off-the-shelf software at a certain refinement level.

2.5 Combination of Dependability Mechanisms — Case Study Part II

We will now develop a revised version of the case study introduced in Section 1.2. As before it will be assumed that the target system to be developed is only the fault-tolerant and secure network layer. Producer *PROD*, consumer *CON* and the transmission media are therefore considered to be objects located in the environment, as depicted in Figure 2.3. However, this time a more realistic setting will be assumed: We will use two instances of the transmission media; M will carry application data and $MACK$ will transmit control messages to acknowledge reception of application data. Both M and $MACK$ are subject to the following assumptions:

1. **Fault hypothesis:** The transmission media may lose messages. The number of consecutive input messages that each medium may lose is bound by $maxLoss \geq 0$.

2. **Security Threat:** A malicious agent X can perform eavesdropping and tamper with messages both on M and $MACK$. Making use of tampering, X can also replay messages by “copying” an old message onto a new package. X cannot block or delay messages or fake the identity of a sender. On each medium, X can modify at most $maxModified \geq 0$ messages in a row.

3. Neither M nor $MACK$ change the ordering of the messages.

The presentation will be structured according to the formal framework given in the previous section. We will first use the trace model of CSP to specify and prove safety properties based on behavioural specifications of type P **sat** $S_P(h)$. After having reached the final level of decomposition, a separate argument will be presented allowing to prove liveness properties by means of automated model checking. I regard this approach as a promising alternative to the simultaneous processing of safety and lifeness properties in the failures-divergence model which is frequently used in CSP applications. The latter technique has been applied by the author in [74] and will be discussed in Chapter 3.

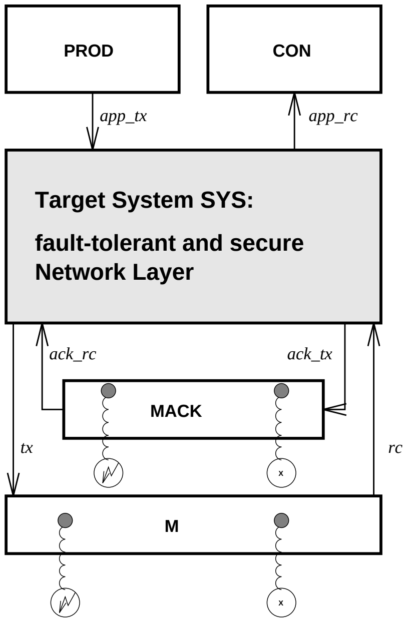

Figure 2.3: System configuration for the case study, part II

2.5.1 System Requirements

Operational Environment

Following the configuration sketched in Figure 2.3 the system context may be specified by

$$\mathcal{E}(SYS) = (PROD \mathbin{|||} CON \mathbin{|||} M \mathbin{|||} MACK) \parallel SYS$$

where the normal behaviour of the environment processes satisfies[7]

E	$\alpha(E)$	E **sat** $S_E(h)$
$PROD$	$\{\| app\_tx \|\}$	$true$
CON	$\{\| app\_rc \|\}$	$true$
M	$\{\| tx, rc \|\}$	$val^*(h \upharpoonright \{\| rc \|\}) \leq^1 val^*(h \upharpoonright \{\| tx \|\})$
$MACK$	$\{\| ack\_tx, ack\_rc \|\}$	$val^*(h \upharpoonright \{\| ack\_rc \|\}) \leq^1 val^*(h \upharpoonright \{\| ack\_tx \|\})$

$\mathcal{E}(SYS)$ introduces an environment architecture where each environment process is completely independent of the others, but operates in parallel with the target system. Therefore the system interface consists of every channel visible in the environment:

$$I = \{app\_tx, app\_rc, tx, rc, ack\_tx, ack\_rc\}$$

Obviously, $ch(S_E(s)) \subseteq ch(E)$ is fulfilled for every specification, since the free trace variable h only appears filtered to the channels of the corresponding processes. Thus the consistency condition for the interface (p. 23) are fulfilled.

[7]For traces s, u, the notation $s \leq^n u$ is used to denote that s is a prefix of u with $\#u - \#s \leq n$.

Application Requirements

The application requirements may then be formally defined as follows[8]:

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

$$\mathcal{E}(SYS)\ \textbf{sat}\ S_{SYS}(h)$$

$$S_{SYS}(h) \equiv_{df} val^*(h \upharpoonright \{\!|\ app\_rc\ |\!\}) \le^N val^*(h \upharpoonright \{\!|\ app\_tx\ |\!\})$$

where $N > 0$ is an unspecified but fixed natural number. $\alpha\,(SYS \setminus L_{SYS})$ is the alphabet of the interface of SYS visible to the environment, where L_{SYS} denotes the set of local SYS-events which will become visible during the decomposition steps to follow.

$S_{SYS}(h)$ requires $(SYS \setminus L_{SYS})$ to input values on *app_tx* and output them on channel *app_rc* without changing their order or value. The constant N in the prefix relation ensures that each output may be expected within a bounded number of consecutive inputs. The fixed upper bound N is appropriate in the context of real-time applications, where the assertion that a message will "finally" reach its destination is not useful at all, because this may still admit failures in the time domain.

Since $I = ch(S_{SYS}(h))$ we are dealing with a black box specification. SYS is obviously underspecified by $S_{SYS}(h)$. For example, it does not show that the transmission media should be used for the purpose of data transfer between $PROD$ and CON. However, this is an architectural issue and will therefore be covered in the system architecture described below. The consistency conditions relating the interface to the channels of SYS (p. 23) are obviously fulfilled.

Threat Analysis (External)

Threat Analysis – Step 1: Definition of Isolated Threats According to the hypotheses informally described above, we are confronted with a reliability threat and a security threat present for both transmission lines M and $MACK$. The security threat can be specified as follows[9]:

$$\begin{array}{l}
\Delta_{M,1}(h, h') \equiv_{df} \\
\quad ch^*(h' \upharpoonright \{\!|\ tx, rc\ |\!\}) = ch^*(h \upharpoonright \{\!|\ tx, rc\ |\!\}) \\
\quad \wedge \\
\quad h' \upharpoonright \{\!|\ tx\ |\!\} = h \upharpoonright \{\!|\ tx\ |\!\} \\
\quad \wedge \\
\quad (\forall\, i : 1 \mathinner{..} (\#(h' \upharpoonright \{\!|\ rc\ |\!\}) - maxModified) \bullet \\
\qquad\qquad (\exists\, j : i \mathinner{..} (i + maxModified) \bullet (h' \upharpoonright \{\!|\ rc\ |\!\})(j) = (h \upharpoonright \{\!|\ rc\ |\!\})(i))) \\
\quad \wedge \\
\quad (\mathrm{ran}\ val^*(h' \upharpoonright \{\!|\ tx\ |\!\}) \subseteq \mathrm{ran}\ \mathsf{e} \Rightarrow \\
\qquad\qquad (\forall\, x : \mathrm{ran}\ val^*(h' \upharpoonright \{\!|\ rc\ |\!\}) \bullet x \notin \mathrm{ran}\,\mathsf{e} \vee x \in \mathrm{ran}\ val^*(h' \upharpoonright \{\!|\ tx\ |\!\}))
\end{array}$$

[8] We use the notation $val^*(h)$ to denote the trace derived from h by projecting each $h(i)$ onto its value component $val(h(i))$.

[9] $ch^*(h)$ denotes the trace derived from h by projecting each $h(i)$ onto its channel $ch(h(i))$. ran h is the set of events contained in h.

In presence of the isolated security threat, M will allow the same inputs as in the case of normal behaviour and preserve the number of outputs, as well as the ordering between inputs and outputs. However, out of $maxModified + 1$ outputs only one can be guaranteed to be unmodified. The last term of the conjunction is a security hypothesis expressing the possibilities to detect modified messages by means of encryption. e denotes an encryption function, whose existence is assumed in the hypothesis, such that we may depend on the following properties: If the values on channel tx are encrypted by means of e (i. e., every $(h \restriction \{\mid tx \mid\})$-value is contained in the range of e), then a malicious agent only has two possibilities to modify the messages, before they reach channel rc. Either a replay is performed (i. e., the modified value is contained in the set of messages already transmitted), or the modified value is not contained in the range of e.

Given any encryption function f, e may be implemented by enciphering the message plus a checksum by means of f: If it is safe to assume that agents have no possibility to decipher the message and encipher an altered message with corresponding checksum, then it is also safe to assume that any modification of the enciphered message will be detected by means of the mismatch between modified message and checksum. More sophisticated encryption mechanisms allowing to detect tampered messages are provided by the *public-key mechanisms*, described for example in [95]. In practice, any reasonably secure e will require the distribution of *encryption keys* between producer and consumer. A formal treatment of key distribution protocols is given in [9].

According to the definition of threats, $\Delta_{M,1}(h, h')$ characterises *acceptable behaviour*. This is expressed by the hypotheses about e, assuming that an agent may only tamper with messages to the extent specified by the fourth conjunct in $\Delta_{M,1}(h, h')$. If an agent might "crack" the encryption code for e and alter messages, such that the result would still look like a valid encryption, this would represent *catastrophic behaviour*, and our protocols to be introduced and verified below would produce unpredictable results.

An analogous threat $\Delta_{MACK,1}(h, h')$ is defined for $MACK$.

The second threat present for M is the hypothesis assuming the possibility of data losses[10]:

$$
\begin{array}{l}
\Delta_{M,2}(h, h') \equiv_{df} \\
\quad h' \restriction \{\mid tx, rc \mid\} \trianglelefteq h \restriction \{\mid tx, rc \mid\} \\
\quad \wedge \\
\quad h' \restriction \{\mid tx \mid\} = h \restriction \{\mid tx \mid\} \\
\quad \wedge \\
\quad (\#(h \restriction \{\mid rc \mid\}) > maxLoss \Rightarrow \\
\qquad \#(h' \restriction \{\mid rc \mid\}) > 0 \\
\qquad \wedge \\
\qquad (\exists\, h^0, h^1, \ldots, h^{\#(h' \restriction \{\mid rc \mid\})} : \{\mid rc \mid\}^* \bullet \\
\qquad\quad \#h^0 \leq maxLoss \wedge h^0 \frown \ldots \frown h^{\#(h' \restriction \{\mid rc \mid\})} = h \restriction \{\mid rc \mid\} \\
\qquad\quad \wedge \\
\qquad\quad (\forall\, i : 1 \,..\, \#(h' \restriction \{\mid rc \mid\}) \bullet \\
\qquad\qquad (h' \restriction \{\mid rc \mid\})(i) = head(h^i) \wedge \#h^i \leq maxLoss + 1)))
\end{array}
$$

[10] $h' \trianglelefteq h$ denotes the *subtrace* relation: Every h'-event is contained in h, and the ordering of h' is also preserved in h.

Again, h' allows the same inputs on $\{\!|\ tx\ |\!\}$ as h, but this time it outputs only a subtrace of $(h \upharpoonright \{\!|\ rc\ |\!\})$. The number of consecutive data losses in limited by $maxLoss$. Let $\Delta_{MACK,2}(h, h')$ be analogously defined for $MACK$. It is trivial to see that $\Delta_{M,j}(h, h')$ and $\Delta_{MACK,j}(h, h')$ are reflexive and prefix-closed, and the full set of consistency conditions (p. 24) for threats holds.

Threat Analysis – Step 2: Compositional Ordering of Threats In the network example a possible ordering of threats can be found along the following lines:

1. If it is reasonable to assume that the malicious agent attacks directly at the input channel tx of M, then the ordering $\langle \Delta_{M,1}, \Delta_{M,2} \rangle$ is justified: *First* a package is possibly corrupted by the agent, *then* it may still be lost by the unreliable network.
2. If it is reasonable to assume that the agent gets a chance to attack directly before M outputs on channel rc, the reverse ordering $\langle \Delta_{M,2}, \Delta_{M,1} \rangle$ is justified: The attacker will now only tamper with packages having already been securely transmitted over the medium.
3. If it cannot be predicted whether the agent may attack at the beginning or at the end of the transmission, it is not possible to make use of the isolated threats $\Delta_{M,1}, \Delta_{M,2}$. If it can be guaranteed, however, that at least $n\%$ of the messages are neither lost nor corrupted, a global threat can be defined as

$$
\begin{aligned}
&\Delta_{M,0}(h, h') \equiv_{df} \\
&\quad ch^*(h' \upharpoonright \{\!|\ tx, rc\ |\!\}) \trianglelefteq ch^*(h \upharpoonright \{\!|\ tx, rc\ |\!\}) \\
&\quad \wedge \\
&\quad h' \upharpoonright \{\!|\ tx\ |\!\} = h \upharpoonright \{\!|\ tx\ |\!\} \\
&\quad \wedge \\
&\quad (\exists\, h'' \bullet (h'' \upharpoonright \{\!|\ tx, rc\ |\!\}) \trianglelefteq (h' \upharpoonright \{\!|\ tx, rc\ |\!\}) \wedge \\
&\qquad\qquad h'' \upharpoonright \{\!|\ tx\ |\!\} = h' \upharpoonright \{\!|\ tx\ |\!\} \wedge \\
&\qquad\qquad \#(h' \upharpoonright \{\!|\ tx\ |\!\}) * n/100 \leq \#(h'' \upharpoonright \{\!|\ rc\ |\!\}) \wedge \\
&\qquad\qquad h'' \upharpoonright \{\!|\ rc\ |\!\} \trianglelefteq h'' \upharpoonright \{\!|\ tx\ |\!\}) \\
&\quad \wedge \\
&\quad (\mathrm{ran}\ val^*(h' \upharpoonright \{\!|\ tx\ |\!\}) \subseteq \mathrm{ran}\ \mathsf{e} \Rightarrow \\
&\qquad\qquad (\forall\, x : \mathrm{ran}\ val^*(h' \upharpoonright \{\!|\ rc\ |\!\}) \bullet x \notin \mathrm{ran}\,\mathsf{e} \vee x \in \mathrm{ran}\ val^*(h' \upharpoonright \{\!|\ tx\ |\!\}))
\end{aligned}
$$

Threat Analysis – Step 3: Derivation of the Resulting Threat Assuming the ordering $\langle \Delta_{M,1}, \Delta_{M,2} \rangle$ of threats results – after having eliminated the bound variable s – in the overall threat

$$
\begin{array}{l}
(\Delta_{M,1} \wr \Delta_{M,2})(h, h') \equiv_{df} \\
\quad ch^*(h' \upharpoonright \{\!\mid tx, rc \mid\!\}) \ \trianglelefteq \ ch^*(h \upharpoonright \{\!\mid tx, rc \mid\!\}) \\
\quad \wedge \\
\quad h' \upharpoonright \{\!\mid tx \mid\!\} = h \upharpoonright \{\!\mid tx \mid\!\} \\
\quad \wedge \\
\quad (\#(h \upharpoonright \{\!\mid rc \mid\!\}) > maxLoss \Rightarrow \\
\qquad \#(h' \upharpoonright \{\!\mid rc \mid\!\}) > 0 \wedge \\
\qquad (\exists\, h^0, h^1, \ldots, h^{\#(h' \upharpoonright \{\!\mid rc \mid\!\})} : \{\!\mid rc \mid\!\}^* \bullet \#h^0 \leq maxLoss \wedge \\
\qquad\quad (\forall\, i : 1 \mathinner{..} (\#(h \upharpoonright \{\!\mid rc \mid\!\}) - maxModified) \bullet \\
\qquad\qquad (\exists\, j : i \mathinner{..} (i + maxModified) \bullet \\
\qquad\qquad\quad (h \upharpoonright \{\!\mid rc \mid\!\})(j) = (h^0 \frown \ldots \frown h^{\#(h' \upharpoonright \{\!\mid rc \mid\!\})})(j)) \wedge \\
\qquad\qquad (\forall\, k : 1 \mathinner{..} \#(h' \upharpoonright \{\!\mid rc \mid\!\}) \bullet \\
\qquad\qquad\quad (h' \upharpoonright \{\!\mid rc \mid\!\})(k) = head(h^k) \wedge \#h^k \leq maxLoss + 1)))) \\
\quad \wedge \\
\quad (\mathrm{ran}\ val^*(h' \upharpoonright \{\!\mid tx \mid\!\}) \subseteq \mathrm{ran}\ \mathsf{e} \Rightarrow \\
\qquad\quad (\forall\, x : \mathrm{ran}\ val^*(h' \upharpoonright \{\!\mid rc \mid\!\}) \bullet x \notin \mathrm{ran}\, \mathsf{e} \vee x \in \mathrm{ran}\ val^*(h' \upharpoonright \{\!\mid tx \mid\!\}))
\end{array}
$$

For $maxModified > 0 \wedge maxLoss > 0$, this predicate admits traces h' containing only modified values on output channel $\{\!\mid rc \mid\!\}$: Choose j such that $(h \upharpoonright \{\!\mid rc \mid\!\})(j)$ never coincides with $head(h^k)$. As a consequence it is impossible to construct a fault-tolerant and secure network service for the medium under the hypothesis $(\Delta_{M,1} \wr \Delta_{M,2})$[11].

Since $\wr$ reflects the compositional ordering of threats, it is not symmetric. Indeed, assuming the ordering $\langle \Delta_{M,2}, \Delta_{M,1} \rangle$, the overall threat derived is

$$
\begin{array}{l}
(\Delta_{M,2} \wr \Delta_{M,1})(h, h') \equiv_{df} \\
\quad ch^*(h' \upharpoonright \{\!\mid tx, rc \mid\!\}) \ \trianglelefteq \ ch^*(h \upharpoonright \{\!\mid tx, rc \mid\!\}) \\
\quad \wedge \\
\quad h' \upharpoonright \{\!\mid tx \mid\!\} = h \upharpoonright \{\!\mid tx \mid\!\} \\
\quad \wedge \\
\quad (\#(h \upharpoonright \{\!\mid rc \mid\!\}) > maxLoss \Rightarrow \\
\qquad \#(h' \upharpoonright \{\!\mid rc \mid\!\}) > 0 \wedge \\
\qquad (\exists\, h^0, h^1, \ldots, h^{\#(h' \upharpoonright \{\!\mid rc \mid\!\})} : \{\!\mid rc \mid\!\}^* \bullet \#h^0 \leq maxLoss \wedge \\
\qquad\quad (\forall\, k : 1 \mathinner{..} \#(h' \upharpoonright \{\!\mid rc \mid\!\}) \bullet \#h^k \leq maxLoss + 1) \wedge \\
\qquad\quad h^0 \frown \ldots \frown h^{\#(h' \upharpoonright \{\!\mid rc \mid\!\})} = h \upharpoonright \{\!\mid rc \mid\!\} \wedge \\
\qquad\quad (\forall\, i : 1 \mathinner{..} (\#(h' \upharpoonright \{\!\mid rc \mid\!\}) - maxModified) \bullet \\
\qquad\qquad (\exists\, j : i \mathinner{..} i + maxModified \bullet (h' \upharpoonright \{\!\mid rc \mid\!\})(j) = head(h^j))))) \\
\quad \wedge \\
\quad (\mathrm{ran}\ val^*(h' \upharpoonright \{\!\mid tx \mid\!\}) \subseteq \mathrm{ran}\ \mathsf{e} \Rightarrow \\
\qquad\quad (\forall\, x : \mathrm{ran}\ val^*(h' \upharpoonright \{\!\mid rc \mid\!\}) \bullet x \notin \mathrm{ran}\, \mathsf{e} \vee x \in \mathrm{ran}\ val^*(h' \upharpoonright \{\!\mid tx \mid\!\}))
\end{array}
$$

If this ordering of threats is realistic, it is possible to extract valid data from the transmission medium, if it does not block completely: The predicate states that the length of $h' \upharpoonright \{\!\mid rc \mid\!\}$ grows with the number of inputs on tx:

[11] If we weaken $\Delta_{M,1}$ such that only every nth message can be modified and $maxLoss < n - 1$, a fault-tolerant and secure network service would exist.

Lemma 1 *Let h, h' be traces such that*

$$(\Delta_{M,2} \wr \Delta_{M,1})(h, h') \wedge 0 \leq \#(h \upharpoonright \{\!|\ tx\ |\!\}) - \#(h \upharpoonright \{\!|\ rc\ |\!\}) \leq 1$$

Then

$$\tfrac{\#(h' \upharpoonright \{\!| tx |\!\}) - maxLoss - 1}{maxLoss + 1} \leq \#(h' \upharpoonright \{\!|\ rc\ |\!\})$$

holds.

Proof.
$0 \leq \#(h \upharpoonright \{\!|\ tx\ |\!\}) - \#(h \upharpoonright \{\!|\ rc\ |\!\}) \leq 1$ ensures that after $maxLoss + 2$ inputs $\#(h \upharpoonright \{\!|\ rc\ |\!\}) > maxLoss$ is fulfilled. Therefore the consequence in the third conjunct of $(\Delta_{M,2} \wr \Delta_{M,1})(h, h')$ holds, so that we may assume the existence of h^i such that

$$h^0 \frown \ldots \frown h^{\#(h' \upharpoonright \{\!| rc |\!\})} = h \upharpoonright \{\!|\ rc\ |\!\}$$

Since $h' \upharpoonright \{\!|\ tx\ |\!\} = h \upharpoonright \{\!|\ tx\ |\!\}$ by the second conjunct of $(\Delta_{M,2} \wr \Delta_{M,1})(h, h')$,
$0 \leq \#(h \upharpoonright \{\!|\ tx\ |\!\}) - \#(h \upharpoonright \{\!|\ rc\ |\!\}) \leq 1$ implies $\#(h' \upharpoonright \{\!|\ tx\ |\!\}) - 1 \leq \#h \upharpoonright \{\!|\ rc\ |\!\})$, so

$$\#(h' \upharpoonright \{\!|\ tx\ |\!\}) - 1 \leq \#(h^0 \frown \ldots \frown h^{\#(h' \upharpoonright \{\!| rc |\!\})})$$

follows. Since $\#h^0 \leq maxLoss$ and $\#h^k \leq maxLoss + 1$ for $i > 0$, we get

$$\#(h' \upharpoonright \{\!|\ tx\ |\!\}) - 1 \leq maxLoss + \#(h' \upharpoonright \{\!|\ rc\ |\!\}) * (maxLoss + 1)$$

As a consequence

$$\tfrac{\#(h' \upharpoonright \{\!| tx |\!\}) - maxLoss - 1}{maxLoss + 1} \leq \#(h' \upharpoonright \{\!|\ rc\ |\!\})$$

□

As soon as $\#(h' \upharpoonright \{\!|\ rc\ |\!\}) > maxModified$, it will be possible to find at least one valid data package out of $maxModified + 1$ consecutive values received on rc. For the remaining part of the case study it is therefore assumed that $\langle \Delta_{M,2}, \Delta_{M,1} \rangle$ is the appropriate compositional ordering of threats.

This example elucidates the fact that design aspects have to be taken into account when calculating the effect of possible faults and security threats. It is a fundamental difference whether malicious agents can only tamper with data successfully transmitted by the medium or whether they can modify data packages while it is still uncertain if a package will reach its destination.

Threat Analysis – Step 4: Derivation of Specifications in Presence of Threats
Using the specification $S_M(h)$ of M given in the operational environment specification (p. 33)

and assuming that the overall threat to M is represented by $(\Delta_{M,2} \wr \Delta_{M,1})$ as derived in the previous example, application of the threat introduction rule and of Lemma 1 results in

$$(M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \ \mathbf{sat} \ (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h')$$

$$\begin{array}{l}
(S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h') \equiv \\
\quad (\exists\, s \bullet val^*(s \restriction \{\!\mid rc \mid\!\}) \leq^1 val^*(s \restriction \{\!\mid tx \mid\!\}) \\
\qquad\qquad \wedge\ ch^*(h' \restriction \{\!\mid tx, rc \mid\!\}) \trianglelefteq ch^*(s \restriction \{\!\mid tx, rc \mid\!\}) \wedge (h' \restriction \{\!\mid tx \mid\!\}) = (s \restriction \{\!\mid tx \mid\!\})) \\
\quad \wedge \\
\quad (\exists\, h'' \bullet (h'' \restriction \{\!\mid tx, rc \mid\!\}) \trianglelefteq (h' \restriction \{\!\mid tx, rc \mid\!\}) \wedge val^*(h'' \restriction \{\!\mid rc \mid\!\}) \leq val^*(h'' \restriction \{\!\mid tx \mid\!\}) \\
\qquad\qquad \wedge\ \frac{\#(h' \restriction \{\!\mid tx \mid\!\}) - maxLoss - 1}{(maxModified + 1) * (maxLoss + 1)} \leq \#(h'' \restriction \{\!\mid rc \mid\!\})) \\
\quad \wedge \\
\quad (\mathrm{ran}\ val^*(h' \restriction \{\!\mid tx \mid\!\}) \subseteq \mathrm{ran}\ \mathsf{e} \Rightarrow \\
\qquad\qquad (\forall\, x : \mathrm{ran}\ val^*(h' \restriction \{\!\mid rc \mid\!\}) \bullet x \notin \mathrm{ran}\, \mathsf{e} \vee x \in \mathrm{ran}\ val^*(h' \restriction \{\!\mid tx \mid\!\}))
\end{array}$$

In this specification the bound variable s represents the normal behaviour trace. The subtrace h'' of the acceptable behaviour trace h' contains the messages that are neither lost nor corrupted. This means that $(M \wr (\Delta_{M,2} \wr \Delta_{M,1}))$ will transmit at least a subtrace of inputs correctly. The behavioural specification in presence of threats is completely analogous for $MACK$.

Dependability Requirements

For SYS it is not allowed that irregularities become visible at the interface to $PROD$ and CON, therefore the dependability requirements say

$$\Delta_{SYS}(h, h') \equiv_{df} h' \restriction \{\!\mid app\_tx, app\_rc \mid\!\} = h \restriction \{\!\mid app\_tx, app\_rc \mid\!\}$$

Since the specification $S_{SYS}(h)$ of normal behaviour only refers to channels app_tx, app_rc, application of the threat introduction rule results in the same assertion:

$$(S_{SYS} \wr \Delta_{SYS})(h') \equiv val^*(h' \restriction \{\!\mid app\_rc \mid\!\}) \leq^N val^*(h' \restriction \{\!\mid app\_tx \mid\!\})$$

2.5.2 System Architecture

System Structure

For the reliable and secure network service we choose a layered architecture

$$SYS = \mathcal{A}_{SYS}(ABP, SEC) = (ABP \parallel SEC)$$

as depicted in Figure 2.4.

This choice is inspired by the observation that the specification of $(M \wr (\Delta_{M,2} \wr \Delta_{M,1}))$ implies the existence of a subtrace $h''' \trianglelefteq h'$ satisfying $(h''' \restriction \{\!\mid tx \mid\!\}) = (s \restriction \{\!\mid tx \mid\!\}) \wedge h''' \trianglelefteq s$, where s is the trace denoting correct behaviour. This type of problem can be solved by means of the alternating bit protocol, which we intend to implement in the ABP layer. To make use of this protocol, we first have to extract h''' from h'. This will be the task of the SEC layer. The allocation of parts of the layers on different computers is deferred to a further decomposition step.

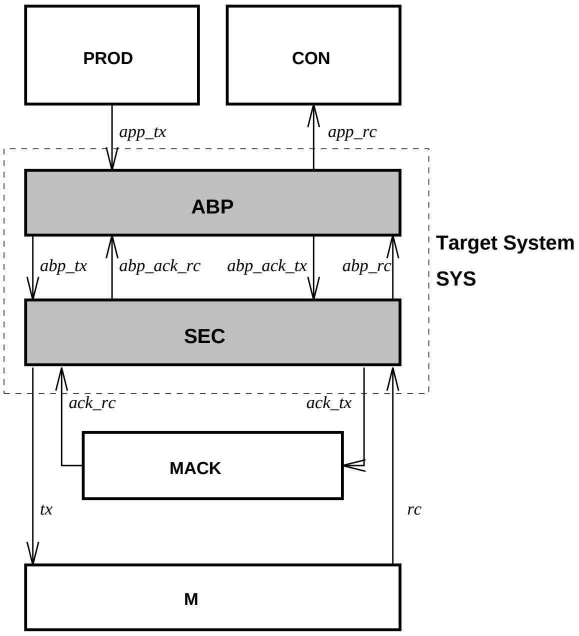

Figure 2.4: System architecture consisting of network layer and security layer.

Interfaces

For the architecture of SYS introduced above we have alphabets

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

$$\alpha(SEC \setminus L_{SEC}) = \{\!|\ tx, rc, ack\_tx, ack\_rc, abp\_tx, abp\_rc, abp\_ack\_tx, abp\_ack\_rc\ |\!\}$$

The interface definition is completed by describing the channel alphabets: Let $DATA$ denote the set of application data to be passed from $PROD$ to CON. Define $BIT = \{0, 1\}$.

Channel	Channel Alphabet
app_tx, app_rc	$DATA$
abp_tx, abp_rc	$BIT \times DATA$
abp_ack_tx, abp_ack_rc	BIT
tx, rc, ack_tx, ack_rc	BIT^*

The channel alphabets of tx and rc are defined to contain “raw data”, i. e., each message on tx, rc is just an uninterpreted sequences of bits. Since the transmission media establish a point-to-point connection, we did not introduce a header data structure, which would precede the data block on a real network.

Obviously, the system interface I is contained in $ch(ABP) \cup ch(SEC)$.

Component Specifications

For the new component ABP we choose the following specification for the alternating bit protocol (see [104, pp. 44]):

$$ABP \textbf{ sat } S_{ABP}(h)$$

$$\begin{array}{l} S_{ABP}(h) \equiv_{df} \\ \quad data^*(\rho(h \upharpoonright \{\!|\ abp\_tx\ |\!\})) \leq^1 val^*(h \upharpoonright \{\!|\ app\_tx\ |\!\}) \\ \quad \wedge \\ \quad val^*(\rho(h \upharpoonright \{\!|\ abp\_ack\_rc\ |\!\})) \leq^1 bit^*(\rho(h \upharpoonright \{\!|\ abp\_tx\ |\!\})) \\ \quad \wedge \\ \quad val^*(h \upharpoonright \{\!|\ app\_rc\ |\!\}) \leq^1 data^*(\rho(h \upharpoonright \{\!|\ abp\_rc\ |\!\})) \\ \quad \wedge \\ \quad val^*(\rho(h \upharpoonright \{\!|\ abp\_ack\_tx\ |\!\})) \leq^1 bit^*(\rho(h \upharpoonright \{\!|\ abp\_rc\ |\!\})) \end{array}$$

In this definition, *bit* and *data* are the projections onto the bit and data component, respectively, of channel events of *abp_tx* and *abp_rc*. For channels c with alphabet *BIT* we may also write $bit(c.b)$ instead of $val(c.b)$ to extract the bit value from a channel event. ρ is a filter for traces deleting duplicated messages identified by means of the bit value on channels with alphabet *BIT* or $BIT \times DATA$:

$$\rho(\langle\ \rangle) = \langle\ \rangle$$

$$\rho(s ^\frown \langle e \rangle) = \textbf{if } \#s > 0 \wedge bit(e) = bit(last(s)) \textbf{ then } \rho(s) \textbf{ else } \rho(s) ^\frown \langle e \rangle$$

For *SEC* we specify

$$SEC \textbf{ sat } S_{SEC}(h)$$

$$\begin{array}{l} S_{SEC}(h) \equiv_{df} \\ \quad val^*(h \upharpoonright \{\!|\ tx\ |\!\}) \leq^1 \epsilon(val^*(h \upharpoonright \{\!|\ abp\_tx\ |\!\})) \\ \quad \wedge \\ \quad val^*(h \upharpoonright \{\!|\ ack\_tx\ |\!\}) \leq^1 \epsilon(val^*(h \upharpoonright \{\!|\ abp\_ack\_tx\ |\!\})) \\ \quad \wedge \\ \quad val^*(h \upharpoonright \{\!|\ abp\_rc\ |\!\}) \leq^1 data^*(\varphi(val^*(h \upharpoonright \{\!|\ rc\ |\!\}))) \\ \quad \wedge \\ \quad val^*(h \upharpoonright \{\!|\ abp\_ack\_rc\ |\!\}) \leq^1 data^*(\varphi(val^*(h \upharpoonright \{\!|\ ack\_rc\ |\!\}))) \end{array}$$

In this specification, ϵ denotes an encryption filter adding a sequence number to each value of a trace and encrypting this pair by means of e:

$$\epsilon(\langle\ \rangle) = \langle\ \rangle$$

$$\epsilon(s ^\frown \langle x \rangle) = \epsilon(s) ^\frown \langle \mathsf{e}(x, \#s + 1) \rangle$$

The filter φ will discard every message which is not correctly encrypted ($x \notin \operatorname{ran} \mathsf{e}$) or which is a replay of an older message, as can be detected by means of the sequence number $snum(\mathsf{e}^{-1}(x))$ of the decoded message:

$$\varphi(\langle\ \rangle) = \langle\ \rangle$$

$$\begin{array}{rl} \varphi(s ^\frown \langle x \rangle) = & \textbf{if } x \notin \operatorname{ran} \mathsf{e} \vee (\#\varphi(s) > 0 \wedge snum(\mathsf{e}^{-1}(x)) \leq snum(last(\varphi(s)))) \\ & \textbf{then } \varphi(s) \\ & \textbf{else } \varphi(s) ^\frown \langle \mathsf{e}^{-1}(x) \rangle \end{array}$$

If the message is accepted by φ the filter outputs its decoded value $\mathsf{e}^{-1}(x)$, consisting of the pair $(data, sequence\_number)$.

Threat Analysis (Internal)

Let us assume that for the architecture $\mathcal{A}_{SYS}(ABP, SEC)$ the possibility of data losses on channel *abp_rc* has to be taken into account (Figure 2.5). As before it is assumed that only $maxLoss \geq 0$ messages may be lost in a row. Formally speaking, there exists a threat

$$
\begin{array}{l}
\Delta_{SEC,1}(h, h') \equiv_{df} \\
\quad (h' \upharpoonright \alpha(SEC)) \trianglelefteq (h \upharpoonright \alpha(SEC)) \\
\quad \wedge \\
\quad (h' \upharpoonright (\alpha(SEC) - \{\!|\ abp\_rc\ |\!\})) = (h \upharpoonright (\alpha(SEC) - \{\!|\ abp\_rc\ |\!\})) \\
\quad \wedge \\
\quad (\#(h \upharpoonright \{\!|\ abp\_rc\ |\!\}) > maxLoss \Rightarrow \\
\qquad \#(h' \upharpoonright \{\!|\ abp\_rc\ |\!\}) > 0 \\
\qquad \wedge \\
\qquad (\exists\, h^0, h^1, \ldots, h^{\#(h' \upharpoonright \{\!|\ abp\_rc |\!\})} : \{\!|\ rc\ |\!\}^* \bullet \\
\qquad\quad \#h^0 \leq maxLoss \wedge h^0 \frown \ldots \frown h^{\#(h' \upharpoonright \{\!|\ abp\_rc |\!\})} = h \upharpoonright \{\!|\ abp\_rc\ |\!\} \\
\qquad\quad \wedge \\
\qquad\quad (\forall\, i : 1 \mathinner{..} \#(h' \upharpoonright \{\!|\ abp\_rc\ |\!\}) \bullet \\
\qquad\qquad (h' \upharpoonright \{\!|\ abp\_rc\ |\!\})(i) = head(h^i) \wedge \#h^i \leq maxLoss + 1)))
\end{array}
$$

Deriving the specification for *SEC* in presence of this threat uses a lemma analogous to Lemma 1 and results in

$$(SEC \wr \Delta_{SEC,1}) \ \mathbf{sat}\ (S_{SEC} \wr \Delta_{SEC,1})(h')$$

$$
\begin{array}{l}
(S_{SEC} \wr \Delta_{SEC,1})(h') \equiv_{df} \\
\quad val^*(h' \upharpoonright \{\!|\ tx\ |\!\}) \leq^1 \epsilon(val^*(h' \upharpoonright \{\!|\ abp\_tx\ |\!\})) \\
\quad \wedge \\
\quad val^*(h' \upharpoonright \{\!|\ ack\_tx\ |\!\}) \leq^1 \epsilon(val^*(h' \upharpoonright \{\!|\ abp\_ack\_tx\ |\!\})) \\
\quad \wedge \\
\quad val^*(h' \upharpoonright \{\!|\ abp\_ack\_rc\ |\!\}) \leq^1 data^*(\varphi(val^*(h' \upharpoonright \{\!|\ ack\_rc\ |\!\}))) \\
\quad \wedge \\
\quad (\exists\, s \bullet val^*(s \upharpoonright \{\!|\ abp\_rc\ |\!\}) \leq^1 data^*(\varphi(val^*(s \upharpoonright \{\!|\ rc\ |\!\}))) \wedge \\
\qquad\qquad h' \upharpoonright \{\!|\ rc, abp\_rc\ |\!\} \trianglelefteq s \upharpoonright \{\!|\ rc, abp\_rc\ |\!\}) \wedge h' \upharpoonright \{\!|\ rc\ |\!\} = s \upharpoonright \{\!|\ rc\ |\!\}) \\
\quad \wedge \\
\quad \frac{\#(\varphi(val^*(h' \upharpoonright \{\!| rc |\!\}))) - maxLoss - 1}{maxLoss + 1} \leq \#(h' \upharpoonright \{\!|\ abp\_rc\ |\!\})
\end{array}
$$

We will see below that $\mathcal{A}_{SYS}(ABP, SEC)$ correctly implements the system requirements, even in the presence of threat $\Delta_{SEC,1}(h, h')$. However, if also a security threat allowing to modify data on channel *abp_tx* had to be taken into account, it is easy to see that the strength of the mechanisms pre-planned for *ABP* and *SEC* would not suffice to provide a dependable network service. An idea to re-design the architecture could be to make the channel *abp_tx* "invulnerable" to agents located inside *SYS*. For example, the transmission component in *SEC* could be linked into the transmission component of *ABP*, so that the communication

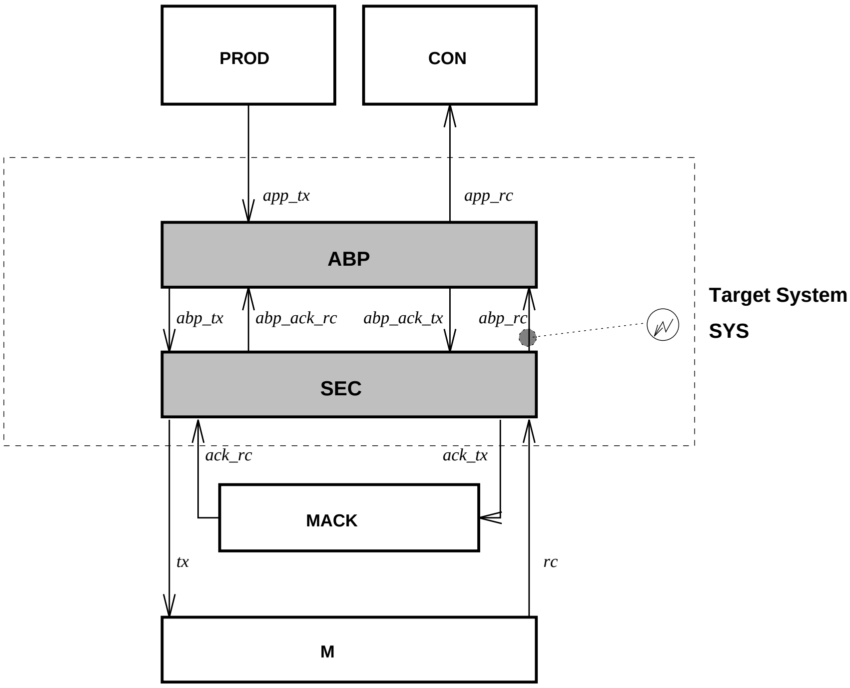

Figure 2.5: Internal threat analysis: Possible loss of data on channel *abp_rc*.

on *abp_tx* could be implemented by means of a procedure call using only local variables not accessible to other processes than ABP. Formally this architecture decision could be modelled as

$$\mathcal{A}'_{SYS}(ABP, SEC) = (ABP \parallel SEC) \setminus \{\!|\ abp\_tx\ |\!\}$$

2.5.3 Verification on System Level

Applying the System Verification Obligations introduced in Section 2.4.4 to the example and observing that the specifications of $PROD$ and CON are just $true$, it has to be shown that the following properties can be derived:

1. System Verification Obligation – Normal Behaviour:

$$\frac{\begin{array}{l} ABP \textbf{ sat } S_{ABP}(h) \\ SEC \textbf{ sat } S_{SEC}(h) \\ M \textbf{ sat } S_M(h) \\ MACK \textbf{ sat } S_{MACK}(h) \end{array}}{(M \mid\mid\mid MACK) \parallel ABP \parallel SEC \textbf{ sat } S(h)}$$

2. System Verification Obligation – Acceptable Behaviour:

$$\frac{\begin{array}{l} ABP \textbf{ sat } S_{ABP}(h) \\ (SEC \wr \Delta_{SEC,1}) \textbf{ sat } (S_{SEC} \wr \Delta_{SEC,1})(h) \\ (M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \textbf{ sat } (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h) \\ (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1})) \textbf{ sat } (S_{MACK} \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))(h) \end{array}}{((M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \mid\mid\mid (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))) \parallel ABP \parallel (SEC \wr \Delta_{SEC,1}) \textbf{ sat } (S \wr \Delta_{SYS})(h)}$$

We skip the proof of the normal behaviour obligation, because the verification concept will become clear in the treatment of the obligation for acceptable behaviour. For the second verification obligation we have assumed that the alternating bit protocol layer is not subject to internal threats. Therefore $S_{ABP}(h)$ and ABP appear without application of the threat introduction operator. We will split this proof obligation into two steps: First it will be derived that the security layer in cooperation with the transmission media will only pass messages to the network layer which have not been corrupted by security attacks, i. e.,

Proof Obligation 1:

$$\frac{\begin{array}{l}(SEC \wr \Delta_{SEC,1})\ \mathbf{sat}\ (S_{SEC} \wr \Delta_{SEC,1})(h)\\ (M \wr (\Delta_{M,2} \wr \Delta_{M,1}))\ \mathbf{sat}\ (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h)\\ (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))\ \mathbf{sat}\ (S_{MACK} \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))(h)\end{array}}{((M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \parallel\!\mid (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))) \parallel (SEC \wr \Delta_{SEC,1})\ \mathbf{sat}\ S_1(h) \wedge S_2(h)}$$

where $S_1(h)$ defines the behaviour of the transmission via M and $S_2(h)$ the behaviour of the transmission via $MACK$:

$$\begin{array}{l} S_1(h) \equiv_{df} \\ \quad val^*(h \upharpoonright \{\!|\ abp\_rc\ |\!\}) \trianglelefteq val^*(h \upharpoonright \{\!|\ abp\_tx\ |\!\}) \\ \quad \wedge \\ \quad \frac{\#(h \upharpoonright \{\!|\ abp\_tx\ |\!\}) - (maxModified+1)*(maxLoss+1)^2 - maxLoss - 2}{(maxModified+1)*(maxLoss+1)^2} \leq \#(h \upharpoonright \{\!|\ abp\_rc\ |\!\}) \end{array}$$

$$\begin{array}{l} S_2(h) \equiv_{df} \\ \quad val^*(h \upharpoonright \{\!|\ abp\_ack\_rc\ |\!\}) \trianglelefteq val^*(h \upharpoonright \{\!|\ abp\_ack\_tx\ |\!\}) \\ \quad \wedge \\ \quad \frac{\#(h \upharpoonright \{\!|\ abp\_ack\_tx\ |\!\}) - (maxModified+1)*(maxLoss+1)^2 - maxLoss - 2}{(maxModified+1)*(maxLoss+1)^2} \leq \#(h \upharpoonright \{\!|\ abp\_ack\_rc\ |\!\}) \end{array}$$

For the second step of the proof we observe that the alternating bit protocol with the behavioural specification as given for ABP above will provide a reliable network service when working in parallel with transmission components satisfying $S_1(h)$ and $S_2(h)$ respectively. We will only give the proof for Proof Obligation 1; a proof for the second step may be found for example in [104, pp. 44].

Lemma 2 *Proof Obligation 1 is implied by*

Proof Obligation 2: *If*

$$(S_{SEC} \wr \Delta_{SEC,1})(h) \wedge (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h)$$

then $S_1(h)$ *holds.*
and
Proof Obligation 3: *If*

$$(S_{SEC} \wr \Delta_{SEC,1})(h) \wedge (S_{MACK} \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))(h)$$

then $S_2(h)$ *holds.*

Proof.
The premises of Proof Obligation 1 state that
$(SEC \wr \Delta_{SEC,1})$ **sat** $(S_{SEC} \wr \Delta_{SEC,1})(h)$, $(M \wr (\Delta_{M,2} \wr \Delta_{M,1}))$ **sat** $(S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h)$
and $(MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))$ **sat** $(S_{MACK} \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))(h)$.

We use the law

$$\frac{P \ \mathbf{sat}\ S_P(h) \qquad Q \ \mathbf{sat}\ S_Q(h)}{(P \parallel Q) \ \mathbf{sat}\ S_P(h \upharpoonright \alpha(P)) \wedge S_Q(h \upharpoonright \alpha(Q))}$$

which holds in the trace model of CSP [44, p. 90]. In our context, the free variable h of a specification P **sat** $S(h)$ appears only in expressions restricted to channels of P, therefore we always have

$$S_P(h \upharpoonright \alpha(P)) \wedge S_Q(h \upharpoonright \alpha(Q)) \Leftrightarrow S_P(h) \wedge S_Q(h)$$

for a trace variable h of $(P \parallel Q)$. Moreover $|||$ is identical to $\parallel$ for disjoint alphabets. It follows that

$$\begin{aligned}&((M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \mathbin{|||} (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))) \parallel (SEC \wr \Delta_{SEC,1}) \ \mathbf{sat} \\ &\quad (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h) \wedge (S_{MACK} \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))(h) \wedge (S_{SEC} \wr \Delta_{SEC,1})(h)\end{aligned}$$

If Proof Obligation 2 is valid, application of the consequence rule results in

$$((M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \mathbin{|||} (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))) \parallel (SEC \wr \Delta_{SEC,1}) \ \mathbf{sat}\ S_1(h)$$

Similarly, if Proof Obligation 3 is valid, application of the consequence rule results in

$$((M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \mathbin{|||} (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))) \parallel (SEC \wr \Delta_{SEC,1}) \ \mathbf{sat}\ S_2(h)$$

Therefore, if both obligations are valid, we may apply the conjunction rule which yields Proof Obligation 1.
□

The following lemmas are used to prepare the proof of Obligation 2.

Lemma 3 *If*

$$(S_{SEC} \wr \Delta_{SEC,1})(h) \wedge (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1})(h)$$

then

$$(\forall x : \mathrm{ran}\ val^*(h \upharpoonright \{\!|\ rc\ |\!\}) \bullet x \notin \mathrm{ran}\,\mathsf{e} \vee x \in \mathrm{ran}\ \epsilon(val^*(h \upharpoonright \{\!|\ abp\_tx\ |\!\})))$$

holds.

Proof.
According to $(S_{SEC} \wr \Delta_{SEC,1})(h)$ (p. 42) and the definition of ϵ (p. 41), every value passed over channel tx is in the range of e. The lemma now follows from the last conjunct in predicate $(S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h)$ (p. 39).
□

The following lemma summarises the implications of specification $(S_{SEC} \wr \Delta_{SEC,1})(h)$ (p. 42) with respect to events observed on channels *abp_rc* and *rc*: An event *abp_rc.y* is always preceded by a communication *rc.x*, such that

- x is correctly encrypted by e.
- y is equal to the data part of the decoded message x.
- The sequence number associated with message x is higher than the sequence number of the most recent previous message on rc which was accepted by the filter φ (p. 41).

Lemma 4 *The validity of* $(S_{SEC} \wr \Delta_{SEC,1})(h)$ *implies*

$$\begin{array}{l}(\forall\, y \bullet \mathit{last}(h \restriction \{\!|\ rc, abp\_rc\ |\!\}) = abp\_rc.y \Rightarrow \\ \quad (\exists\, x \bullet \mathit{last}(\mathit{front}(h \restriction \{\!|\ rc, abp\_rc\ |\!\}))) = rc.x \wedge \\ \qquad x \in \operatorname{ran} \mathsf{e} \wedge \\ \qquad y = \mathit{data}(\mathsf{e}^{-1}(x)) \wedge \\ \qquad (\#(h \restriction \{\!|\ rc\ |\!\}) > 1 \Rightarrow \\ \qquad\quad \mathit{snum}(\mathsf{e}^{-1}(x)) > \mathit{snum}(\mathit{last}(\varphi(\mathit{val}^*(\mathit{front}(h \restriction \{\!|\ rc\ |\!\}))))))))\end{array}$$

Proof.
According to the fourth conjunct in $(S_{SEC} \wr \Delta_{SEC,1})(h)$ (p. 42), we may assume the existence of a trace s satisfying

$$\begin{array}{l}\mathit{val}^*(s \restriction \{\!|\ abp\_rc\ |\!\}) \le^1 \mathit{data}^*(\varphi(\mathit{val}^*(s \restriction \{\!|\ rc\ |\!\}))) \wedge \\ \qquad\qquad h \restriction \{\!|\ rc, abp\_rc\ |\!\} \trianglelefteq s \restriction \{\!|\ rc, abp\_rc\ |\!\}) \wedge h \restriction \{\!|\ rc\ |\!\} = s \restriction \{\!|\ rc\ |\!\}\end{array}$$

Assume that the lemma has been proven for those traces h satisfying $h = s$, i. e., for those h where data losses on abp_rc did not occur. Then the lemma will also hold for arbitrary h satisfying $(S_{SEC} \wr \Delta_{SEC,1})(h)$, because $h \restriction \{\!|\ rc, abp\_rc\ |\!\} \trianglelefteq s \restriction \{\!|\ rc, abp\_rc\ |\!\}) \wedge h \restriction \{\!|\ rc\ |\!\} = s \restriction \{\!|\ rc\ |\!\}$, and the implication of the lemma remains valid for any trace derived from s by deleting abp_rc-events only.

For rest of the proof we may therefore assume

$$\mathit{val}^*(h \restriction \{\!|\ abp\_rc\ |\!\}) \le^1 \mathit{data}^*(\varphi(\mathit{val}^*(h \restriction \{\!|\ rc\ |\!\}))) \tag{0}$$

Let $(h \restriction \{\!|\ rc, abp\_rc\ |\!\}) = (u \restriction \{\!|\ rc, abp\_rc\ |\!\}) ^\frown \langle abp\_rc.y \rangle$. Then $(u \restriction \{\!|\ rc, abp\_rc\ |\!\})$ cannot be empty because of (0).

We will first prove that $ch(\mathit{last}(u \restriction \{\!|\ rc, abp\_rc\ |\!\})) = rc$ by assuming the contrary and deriving a contradiction. Assume therefore that $ch(\mathit{last}(u \restriction \{\!|\ rc, abp\_rc\ |\!\})) = abp\_rc$. Then

$$\#(h \restriction \{\!|\ abp\_rc\ |\!\}) = \#(\mathit{front}(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ abp\_rc\ |\!\}) + 2 \tag{1}$$

and

$$h \restriction \{\!|\ rc\ |\!\} = \mathit{front}(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ rc\ |\!\} \tag{2}$$

Now (0) together with (2) implies

$$\mathit{val}^*(h \restriction \{\!|\ abp\_rc\ |\!\}) \le^1 \mathit{data}^*(\varphi(\mathit{val}^*(\mathit{front}(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ rc\ |\!\}))))$$

and therefore

$$\#(h \restriction \{\!|\ abp\_rc\ |\!\}) \le \#(\varphi(\mathit{val}^*(\mathit{front}(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ rc\ |\!\}))) \tag{3}$$

Since $u \leq h$ and (0) is also valid for every prefix of h, this yields

$$val^*(front(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ abp\_rc\ |\!\}))) \leq^1 data^*(\varphi(val^*(front(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ rc\ |\!\}))))$$

and therefore

$$\#(\varphi(val^*(front(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ rc\ |\!\})))) \leq \#(front(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ abp\_rc\ |\!\})) + 1$$

With (3), this results in

$$\#(h \restriction \{\!|\ abp\_rc\ |\!\}) \leq \#(front(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) \restriction \{\!|\ abp\_rc\ |\!\})) + 1$$

which contradicts (1).

Let therefore $last(u \restriction \{\!|\ rc, abp\_rc\ |\!\}) = rc.x$.

Case (C1): Assume $(h \restriction \{\!|\ rc\ |\!\}) = \langle rc.x \rangle$. If $(h \restriction \{\!|\ rc, abp\_rc\ |\!\}) = \langle rc.x, abp\_rc.y \rangle$, (0) implies $val^*(h \restriction \{\!|\ abp\_rc\ |\!\}) = data^*(\varphi(val^*(h \restriction \{\!|\ rc\ |\!\})))$, that is, $\langle y \rangle = data^*(\varphi(\langle x \rangle))$. Now the definition of φ applied to the case $\varphi(\langle\,\rangle ^\frown \langle x \rangle)$ yields $x \in \operatorname{ran} \mathsf{e} \wedge \varphi(\langle x \rangle) = \langle \mathsf{e}^{-1}(x) \rangle$ which proves the lemma for $\#(h \restriction \{\!|\ rc\ |\!\}) = 1$.

Case (C2): Assume $\#(h \restriction \{\!|\ rc\ |\!\}) > 1 \wedge last(h \restriction \{\!|\ rc, abp\_rc\ |\!\}) = abp\_rc.y$. Again, (0) implies $val^*(h \restriction \{\!|\ abp\_rc\ |\!\}) = data^*(\varphi(val^*(h \restriction \{\!|\ rc\ |\!\})))$, which now results in $y = data(last(\varphi(val^*(h \restriction \{\!|\ rc\ |\!\}))))$. From the recursive definition of φ (p. 41) we get (using induction over the length of sequences z)

$$\#\varphi(z) > 0 \Rightarrow last(\varphi(z)) = \mathsf{e}^{-1}(last(z))$$

It follows that

$$last(\varphi(val^*(h \restriction \{\!|\ rc\ |\!\}))) = \mathsf{e}^{-1}(val(last(h \restriction \{\!|\ rc\ |\!\}))) = \mathsf{e}^{-1}(x)$$

and

$$\varphi(val^*(h \restriction \{\!|\ rc\ |\!\})) = \varphi(val^*(front(h \restriction \{\!|\ rc\ |\!\}))) ^\frown \langle \mathsf{e}^{-1}(x) \rangle$$

Now the definition of φ implies

$$snum(\mathsf{e}^{-1}(x)) > snum(last(\varphi(val^*(front(h \restriction \{\!|\ rc\ |\!\})))))$$

which completes the proof.
□

Lemma 5 *If*

$$(S_{SEC} \wr \Delta_{SEC,1})(h) \wedge (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1})(h)$$

holds, this implies

$$\frac{\#(h \restriction \{\!|\ abp\_tx\ |\!\}) - maxLoss - 2}{(maxModified + 1) * (maxLoss + 1)} \leq \#\varphi(val^*(h \restriction \{\!|\ rc\ |\!\}))$$

Proof.
The first conjunct in $(S_{SEC} \wr \Delta_{SEC,1})(h)$ is $val^*(h \restriction \{\!| \ tx \ |\!\}) \leq^1 \epsilon(val^*(h \restriction \{\!| \ abp\_tx \ |\!\}))$. Since $\#\epsilon(s) = \#s$ for every trace s, this yields

$$\#(h \restriction \{\!| \ abp\_tx \ |\!\})) - 1 \leq \#(h \restriction \{\!| \ tx \ |\!\})$$

Together with the second conjunct of $(S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h)$ this implies

$$(\exists\, h'' \bullet (h'' \restriction \{\!| \ tx, rc \ |\!\}) \trianglelefteq (h \restriction \{\!| \ tx, rc \ |\!\}) \wedge val^*(h'' \restriction \{\!| \ rc \ |\!\}) \leq val^*(h'' \restriction \{\!| \ tx \ |\!\}) \wedge \frac{\#(h \restriction \{\!| \ abp\_tx \ |\!\}) - maxLoss - 2}{(maxModified + 1) * (maxLoss + 1)} \leq \#(h'' \restriction \{\!| \ rc \ |\!\}))$$

The lemma will now follow if $\#(h'' \restriction \{\!| \ rc \ |\!\})) \leq \#\varphi(val^*(h \restriction \{\!| \ rc \ |\!\}))$ can be established. To this end, we will show that every $(h'' \restriction \{\!| \ rc \ |\!\})$-event in h adds an entry to $\varphi(val^*(h \restriction \{\!| \ rc \ |\!\}))$. Assume therefore that $last(h \restriction \{\!| \ tx, rc \ |\!\}) = last(h'' \restriction \{\!| \ rc \ |\!\})$. We show that this implies

$$\varphi(val^*(h \restriction \{\!| \ rc \ |\!\})) = \varphi(val^*(front(h \restriction \{\!| \ rc \ |\!\})))^\frown\langle \mathsf{e}^{-1}(val(last(h'' \restriction \{\!| \ rc \ |\!\})))\rangle$$

Since $val^*(h'' \restriction \{\!| \ rc \ |\!\})$ is a subtrace of tx-values, the first conjunct in $(S_{SEC} \wr \Delta_{SEC,1})(h)$ implies that $\operatorname{ran} val^*(h'' \restriction \{\!| \ rc \ |\!\}) \subseteq \operatorname{ran} \mathsf{e}$. If $\#(h \restriction \{\!| \ rc \ |\!\}) = 1$, this means that $val(h \restriction \{\!| \ rc \ |\!\})$ will be accepted by the filter φ. If $\#(h \restriction \{\!| \ rc \ |\!\}) > 1$, the first conjunct of $(S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h)$ implies that $val(last(h'' \restriction \{\!| \ rc \ |\!\}))$ is the value of the most recent tx-event transmitted before, because $M \wr (\Delta_{M,2} \wr \Delta_{M,1})$ acts as an (unreliable) one-place buffer. Since ϵ adds strictly increasing sequence numbers, the third conjunct in $(S_M \wr (\Delta_{M,2} \wr \Delta_{M,1})(h)$ implies that $snum(\mathsf{e}^{-1}(val(last(h'' \restriction \{\!| \ rc \ |\!\}))))$ is greater than any other sequence number of $(h \restriction \{\!| \ rc \ |\!\})$-events. Now the definition of φ implies that $\mathsf{e}^{-1}(val(last(h'' \restriction \{\!| \ rc \ |\!\})))$ is appended to the sequence filtered by φ. This completes the proof.
□

Proof of Obligation 2 : If $(S_{SEC} \wr \Delta_{SEC,1})(h) \wedge (S_M \wr (\Delta_{M,2} \wr \Delta_{M,1}))(h)$ then $S_1(h)$ follows.

To prove $val^*(h \restriction \{\!| \ abp\_rc \ |\!\}) \trianglelefteq val^*(h \restriction \{\!| \ abp\_tx \ |\!\})$, we will use induction over the length of $(h \restriction \{\!| \ rc, abp\_rc \ |\!\})$. Clearly, the assertion holds for $(h \restriction \{\!| \ rc, abp\_rc \ |\!\}) = \langle \ \rangle$. Now assume that it holds for all traces of length less or equal $n \geq 0$ and let $(h \restriction \{\!| \ rc, abp\_rc \ |\!\}) = (u \restriction \{\!| \ rc, abp\_rc \ |\!\})^\frown\langle a \rangle$ with $\#(u \restriction \{\!| \ rc, abp\_rc \ |\!\}) = n$ and some rc- or abp_rc-event a.

Case (C1): Assume that $(u \restriction \{\!| \ rc, abp\_rc \ |\!\}) = \langle \ \rangle$ or $ch(last(u \restriction \{\!| \ rc, abp\_rc \ |\!\})) = abp\_rc$. Then Lemma 4 implies that $(u \restriction \{\!| \ rc, abp\_rc \ |\!\})$ can only be continued by an rc-event, i. e., we may assume $(h \restriction \{\!| \ rc, abp\_rc \ |\!\}) = (u \restriction \{\!| \ rc, abp\_rc \ |\!\})^\frown\langle rc.x \rangle$ for some $x \in \alpha(rc)$. As a consequence

$$\begin{aligned} val^*(h \restriction \{\!| \ abp\_rc \ |\!\}) &= val^*(u \restriction \{\!| \ abp\_rc \ |\!\}) \\ &\trianglelefteq val^*(u \restriction \{\!| \ abp\_tx \ |\!\}) && \text{[induction hypothesis]} \\ &\leq val^*(h \restriction \{\!| \ abp\_tx \ |\!\}) && [u \leq h] \end{aligned}$$

Case (C2): Assume that $last(u \restriction \{\!| \ rc, abp\_rc \ |\!\}) = rc.x$.

Case (C21): Assume that
$x \notin \operatorname{ran} \mathsf{e} \vee snum(\mathsf{e}^{-1}(x)) \leq snum(last(\varphi(val^*(front(u \restriction \{\!| \ rc \ |\!\})))))$

Then according to Lemma 4 a must be an rc-event and we have again $h \restriction \{\!| \ abp\_rc \ |\!\} = u \restriction \{\!| \ abp\_rc \ |\!\}$ as in Case (C1).

Case (C22): Assume that
$x \in \operatorname{ran} \mathsf{e} \wedge snum(\mathsf{e}^{-1}(x)) > snum(last(\varphi(val^*(front(u \restriction \{\!| \ rc \ |\!\})))))$
Then a may be an rc-event ($rc.x$ is lost before an abp_rc-output occurs) or an abp_rc event. In the former case we use the same argument as in (C1). In the latter case we get

$$
\begin{aligned}
val^*(h \restriction \{\!| \ abp\_rc \ |\!\}) &= val^*((u \restriction \{\!| \ abp\_rc \ |\!\})^\frown \langle data(\mathsf{e}^{-1}(x))\rangle) && \text{[Lemma 4]}\\
&\trianglelefteq val^*((u \restriction \{\!| \ abp\_tx \ |\!\})^\frown \langle data(\mathsf{e}^{-1}(x))\rangle) && \text{[induction hypothesis]}\\
&\trianglelefteq val^*(h \restriction \{\!| \ abp\_tx \ |\!\})
\end{aligned}
$$
$$
[\text{Lemma 3, def. of } \epsilon \text{ and } snum(\mathsf{e}^{-1}(x)) > snum(last(\varphi(val^*(front(u \restriction \{\!| \ rc \ |\!\})))))]
$$

This completes the inductive proof of $val^*(h \restriction \{\!| \ abp\_rc \ |\!\}) \trianglelefteq val^*(h \restriction \{\!| \ abp\_tx \ |\!\})$.

For the estimate of $\#(h \restriction \{\!| \ abp\_rc \ |\!\})$ we calculate

$$
\begin{aligned}
\#(h \restriction \{\!| \ abp\_rc \ |\!\}) &\geq \frac{\#(\varphi(val^*(h \restriction \{\!| rc |\!\}))) - maxLoss - 1}{maxLoss+1} && [(S_{SEC} \wr \Delta_{SEC,1})(h)]\\
&\geq \frac{\frac{\#(h \restriction \{\!| abp\_tx |\!\}) - maxLoss - 2}{(maxModified+1)*(maxLoss+1)} - maxLoss - 1}{maxLoss+1} && \text{[Lemma 5]}\\
&= \frac{\#(h \restriction \{\!| abp\_tx |\!\}) - (maxModified+1)*(maxLoss+1)^2 - maxLoss - 2}{(maxModified+1)*(maxLoss+1)^2}
\end{aligned}
$$

This completes the proof of Obligation 2; Obligation 3 is derived in an analogous way.
□

2.5.4 Recursive Application of the Development Procedure

For the architecture $\mathcal{A}_{SYS}(ABP, SEC)$ depicted in Figure 2.4 further decomposition steps may be taken according to Figure 2.6.

For ABP the architecture $\mathcal{A}_{ABP}(ABP\_TX, ABP\_RC) = (ABP\_TX \parallel\!| ABP\_RC)$ is chosen. ABP_TX and ABP_RC have alphabets $\alpha(ABP\_TX) = \{\!| \ app\_tx, abp\_tx, abp\_ack\_rc \ |\!\}$ and $\alpha(ABP\_RC) = \{\!| \ app\_rc, abp\_rc, abp\_ack\_tx \ |\!\}$ and are specified by

$$ABP\_TX \ \textbf{sat} \ S_{ABP\_TX}(h)$$

$$
\begin{aligned}
&S_{ABP\_TX}(h) \equiv_{df}\\
&\quad data^*(\rho(h \restriction \{\!| \ abp\_tx \ |\!\})) \leq^1 val^*(h \restriction \{\!| \ app\_tx \ |\!\})\\
&\quad \wedge\\
&\quad val^*(\rho(h \restriction \{\!| \ abp\_ack\_rc \ |\!\})) \leq^1 bit^*(\rho(h \restriction \{\!| \ abp\_tx \ |\!\}))
\end{aligned}
$$

and

$$ABP\_RC \ \textbf{sat} \ S_{ABP\_RC}(h)$$

$$
\begin{aligned}
&S_{ABP\_RC}(h) \equiv_{df}\\
&\quad val^*(h \restriction \{\!| \ app\_rc \ |\!\}) \leq^1 data^*(\rho(h \restriction \{\!| \ abp\_rc \ |\!\}))\\
&\quad \wedge\\
&\quad val^*(\rho(h \restriction \{\!| \ abp\_ack\_tx \ |\!\})) \leq^1 bit^*(\rho(h \restriction \{\!| \ abp\_rc \ |\!\}))
\end{aligned}
$$

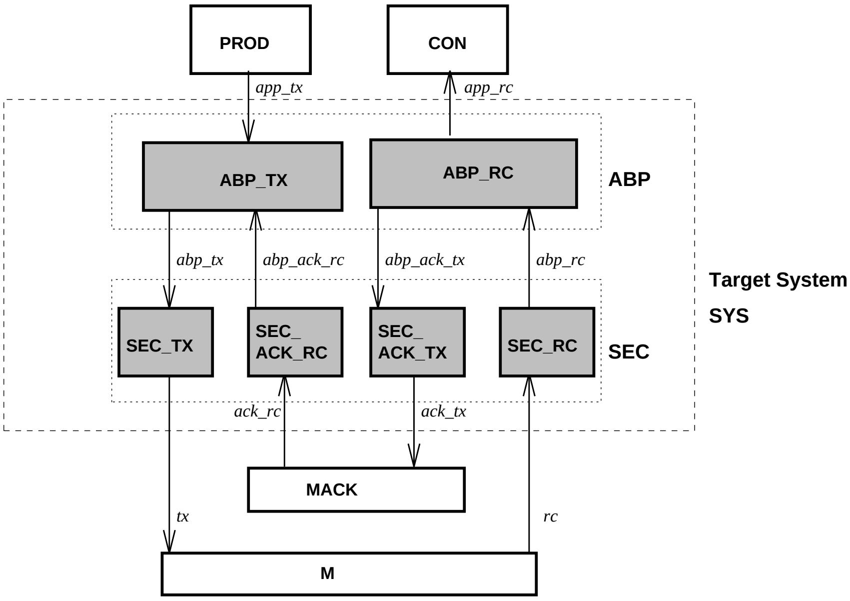

Figure 2.6: Correct combination of a new security layer with re-used network layer.

Since ABP_TX and ABP_RC have disjoint alphabets and are composed by means of interleaving, it is trivial that their specifications imply the component requirements specification $S_{ABP}(h)$ since the latter is just the conjunction of $S_{ABP\_TX}(h)$ and $S_{ABP\_RC}(h)$.

Analogously, SEC is decomposed by

$$\begin{aligned}\mathcal{A}_{SEC}(SEC\_TX, SEC\_RC, SEC\_ACK\_TX, SEC\_ACK\_RC) = \\ (SEC\_TX \mid\mid\mid SEC\_RC \mid\mid\mid SEC\_ACK\_TX \mid\mid\mid SEC\_ACK\_RC)\end{aligned}$$

with the corresponding terms from $S_{SEC}(h)$ as component specifications.

In the next decomposition for each of the components $ABP\_TX, \dots, SEC\_ACK\_RC$ it will be appropriate to introduce *explicit* specifications, because we will reach the level of isolated sequential processes, which are straightforward to implement in the target programming language. The following processes may be used to implement the implicit specifications of the security layer:

$$SEC\_TX = STX(1)$$

$$STX(n) = abp\_tx?x \rightarrow tx!\mathsf{e}(x, n) \rightarrow STX(n+1)$$

$$SEC\_RC = SRC(0)$$

$$\begin{aligned}SRC(n) = rc?z \rightarrow (&\mathbf{if}\ z \notin \operatorname{ran}\mathsf{e} \vee snum(\mathsf{e}^{-1}(z)) \leq n \\ &\mathbf{then}\ SRC(n) \\ &\mathbf{else}\ (abp\_rc!(bit(\mathsf{e}^{-1}(z)), data(\mathsf{e}^{-1}(z))) \rightarrow SRC(n+1)))\end{aligned}$$

SEC_ACK_TX, SEC_ACK_RC are defined analogously. For the network layer we may re-use the processes ABP_TX and ABP_RC defined in Chapter 1 (p. 8).

To prove that these processes correctly implement their corresponding behavioural specifications two techniques may be used:

- Application of CSP laws connecting explicit process algebraic representations to their corresponding implicit specifications, as given by Hoare [44].
- Assertional reasoning about CSP processes in *normal form representation* according to the proof rules introduced by Apt and Olderog [5]. The link between normal form processes and their corresponding behavioural specifications can be realised by means of auxiliary variables representing traces and refusals, as described by the author in [74].

The second approach will be explained in Chapter 3, we will therefore skip the proofs here.

2.5.5 Verification of Deadlock Freedom

The verification in the trace model of CSP performed above may be informally summarised as: "*If* values are delivered on output channels, they will be correct with respect to the system requirements". As a by-product, the verification process also produced estimates for the number of messages to be delivered over output channels. For example, Proof Obligation 2 (p. 44) showed for the security layer that the number of messages delivered via output channel abp_rc grows with the number of inputs on abp_tx. Analogous estimates may be given for channels abp_ack_rc (Proof Obligation 3) and finally for app_rc. These estimates simplify the proof of non-blocking properties: It only has to be shown that the complete system will allow an unbounded number of events on certain input channels. For our case study these channels are app_tx, abp_tx and abp_ack_tx. Formally, the obligation to allow unbounded numbers of events on a channel may be expressed using a refinement relation in the failures-divergence model of CSP. For the case study we have the following proof obligations:

Non-Blocking Proof Obligation 1: For $P_1 = app\_tx?x \rightarrow P_1$ show that

$$P_1 \sqsubseteq_{FD} ESYS \setminus (\alpha(ESYS) - \{\!|\ app\_tx\ |\!\})$$

Non-Blocking Proof Obligation 2: For $P_2 = abp\_tx?x \rightarrow P_2$ show that

$$P_2 \sqsubseteq_{FD} ESYS \setminus (\alpha(ESYS) - \{\!|\ abp\_tx\ |\!\})$$

Non-Blocking Proof Obligation 3: For $P_3 = abp\_ack\_tx?x \rightarrow P_3$ show that

$$P_3 \sqsubseteq_{FD} ESYS \setminus (\alpha(ESYS) - \{\!|\ abp\_ack\_tx\ |\!\})$$

In these obligations, $ESYS$ stands for the completely developed system and its environment with explicit representations of $ABP\_TX, \ldots, SEC\_ACK\_RC$:

$$\begin{aligned} ESYS = {} & (PROD \mathbin{|||} CON) \parallel \\ & (ABP\_TX \mathbin{|||} ABP\_RC) \parallel \\ & (SEC\_TX \mathbin{|||} (SEC\_RC \wr \Delta_{SEC,1}) \mathbin{|||} SEC\_ACK\_TX \mathbin{|||} SEC\_ACK\_RC) \parallel \\ & ((M \wr (\Delta_{M,2} \wr \Delta_{M,1})) \mathbin{|||} (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))) \end{aligned}$$

Since these obligations only refer to the possible *occurrences* of events, but do not require to consider actual data values, it seems promising to try a mechanised proof. To this end, let us follow the technique of *abstract interpretations* for distributed systems (see Clarke *et al.* [16]) and analyse the following processes that represent simplified but less deterministic versions of the corresponding processes $ABP\_TX, \ldots$:

$$
\begin{array}{ll}
ATX_1 & = app\_tx?x \to ATX_{11} \mathbin{\Box} abp\_ack\_rc?y \to ATX_1 \\[1ex]
ATX_{11} & = (\sqcap_{z:\alpha(abp\_tx)} abp\_tx!z \to ATX_{11}) \mathbin{\Box} (abp\_ack\_rc?x \to (ATX_1 \sqcap ATX_{11})) \\[1ex]
ARC_1 & = abp\_rc?x \to \\
& \qquad (\sqcap_{z:\alpha(app\_rc)} app\_rc!z \to (\sqcap_{w:\alpha(abp\_ack\_tx)} abp\_ack\_tx!w \to ARC_1)) \\
& \qquad \sqcap \\
& \qquad (\sqcap_{w:\alpha(bp\_ack\_tx)}) abp\_ack\_tx!w \to ARC_1) \\[1ex]
STX_1 & = abp\_tx?x \to (\sqcap_{y:\alpha(tx)} tx!y \to STX_1) \\[1ex]
SRC_1 & = SRC_1(m_1) \\
SRC_1(n) & = rc?x \to (\mathbf{if}\ (n = 0) \\
& \qquad\qquad \mathbf{then}\ (\sqcap_{y:\alpha(abp\_rc)} abp\_rc!y \to SRC_1(m_1)) \\
& \qquad\qquad \mathbf{else}\ (SRC_1(n-1) \sqcap (\sqcap_{y:\alpha(abp\_rc)} abp\_rc!y \to SRC_1(m_1)))) \\[1ex]
M_1 & = M_1(maxLoss) \\
M_1(n) & = tx?x \to (\mathbf{if}\ (n = 0) \\
& \qquad\qquad \mathbf{then}\ (\sqcap_{y:\alpha(rc)} rc!y \to M_1(maxLoss)) \\
& \qquad\qquad \mathbf{else}\ (M_1(n-1) \sqcap (\sqcap_{y:\alpha(rc)} rc!y \to M_1(maxLoss)))) \\[1ex]
MACK_1 & = MACK_1(maxLoss) \\
MACK_1(n) & = ack\_tx?x \to \\
& \quad (\mathbf{if}\ (n = 0) \\
& \quad\ \ \mathbf{then}\ (\sqcap_{y:\alpha(ack\_rc)} ack\_rc!y \to MACK_1(maxLoss)) \\
& \quad\ \ \mathbf{else}\ (MACK_1(n-1) \sqcap (\sqcap_{y:\alpha(ack\_rc)} ack\_rc!y \to MACK_1(maxLoss))))
\end{array}
$$

In these definitions, m_1 is a constant derived from Lemma 5. It indicates that SRC_1 will not refuse output for an unbounded number of input events.

Obviously, $ATX_1 \sqsubseteq_{FD} ABP\_TX$, because the communication structure of ATX_1 is the same as in ABP_TX, but for ATX_1 the degree of nondeterminism has been increased by replacing internal decisions of ABP_TX by the $\sqcap$-operator. Similarly, $ARC_1, \ldots, MACK_1$ are refined by $ABP\_RC, \ldots, (MACK \wr (\Delta_{MACK,2} \wr \Delta_{MACK,1}))$ in the failures-divergence model. Therefore Non-Blocking Proof Obligations 1 to 3 are implied by the corresponding refinement proofs for system $ESYS_1 = ESYS[ATX_1/ABP\_TX, \ldots]$ against P_1, P_2 and P_3.

Next observe that the behaviour of $ATX_1, \ldots, MACK_1$ does not depend on the values received on any input channel. Therefore a non-blocking property referring only to the occurrence of *any* channel event holds for $ESYS_1$ if and only if it holds for $ESYS_2$ =

$ESYS_1[ATX_2/ATX_1, \ldots]$ with processes

$$
\begin{array}{ll}
ATX_2 & = app\_tx_2 \rightarrow ATX_{21} \,\Box\, abp\_ack\_rc_2 \rightarrow ATX_2 \\[1em]
ATX_{21} & = abp\_tx_2 \rightarrow ATX_{21} \,\Box\, abp\_ack\_rc_2 \rightarrow (ATX_2 \sqcap ATX_{21}) \\[1em]
ARC_2 & = abp\_rc_2 \rightarrow (app\_rc_2 \rightarrow abp\_ack\_rc_2 \rightarrow ARC_2 \\
 & \qquad\qquad\quad \sqcap \\
 & \qquad\qquad\quad abp\_ack\_tx_2 \rightarrow ARC_2) \\[1em]
STX_2 & = abp\_tx_2 \rightarrow tx_2 \rightarrow STX_2 \\[1em]
SRC_2 & = SRC_2(m_1) \\
SRC_2(n) & = rc_2 \rightarrow (\mathbf{if}\ (n = 0) \\
 & \qquad\quad \mathbf{then}\ (abp\_rc_2 \rightarrow SRC_2(m_1)) \\
 & \qquad\quad \mathbf{else}\ (SRC_2(n-1) \sqcap abp\_rc_2 \rightarrow SRC_2(m_1))) \\[1em]
M_2 & = M_2(maxLoss) \\
M_2(n) & = tx_2 \rightarrow (\mathbf{if}\ (n = 0) \\
 & \qquad\quad \mathbf{then}\ (rc_2 \rightarrow M_2(maxLoss)) \\
 & \qquad\quad \mathbf{else}\ (M_2(n-1) \sqcap (rc_2 \rightarrow M_2(maxLoss)))) \\[1em]
MACK_2 & = MACK_2(0) \\
MACK_2(n) & = ack\_tx_2 \rightarrow (\mathbf{if}\ (n = 0) \\
 & \qquad\quad \mathbf{then}\ (ack\_rc_2 \rightarrow MACK_2(maxLoss)) \\
 & \qquad\quad \mathbf{else}\ (MACK_2(n-1) \sqcap ack\_rc_2 \rightarrow MACK_2(maxLoss)))
\end{array}
$$

In these definitions $app\_tx_2, abp\_ack\_tx_2, \ldots$ are no longer channels but single events. The validity of the proof obligations above is now implied by the validity of

Non-Blocking Proof Obligation 1': For $P_1' = app\_tx \rightarrow P_1'$ show that

$$P_1' \sqsubseteq_{FD} ESYS_2 \setminus (\alpha(ESYS_2) - \{app\_tx\})$$

Non-Blocking Proof Obligation 2': For $P_2' = abp\_tx \rightarrow P_2'$ show that

$$P_2' \sqsubseteq_{FD} ESYS_2 \setminus (\alpha(ESYS_2) - \{abp\_tx\})$$

Non-Blocking Proof Obligation 3': For $P_3' = abp\_ack\_tx \rightarrow P_3'$ show that

$$P_3' \sqsubseteq_{FD} ESYS_2 \setminus (\alpha(ESYS_2) - \{abp\_ack\_tx\})$$

System $ESYS_2$ fulfills the requirements for the FDR model checker [27]: every sequential process has only a finite number of states and works with finite alphabets only. Indeed, model checking with FDR shows that the Non-Blocking Proof Obligations 1' to 3' hold for system $ESYS_2$, as was to be shown.

2.6 Discussion and Future Work

In this chapter we have presented a re-usable and in parts standardised framework for the development of dependable systems. Although the framework is based on the *V-Model*, it may be adapted to other standards (for example, [22]) with only minor modifications, like using other names for the documents and slightly different orderings of the items to be described. This global applicability of the framework is an important aspect, because there seems to be little chance for an internationally accepted development standard to emerge from the various standards presently discussed. On the other hand, we do not consider the *formal method* applied in this chapter to be universally applicable. It just served as an example of a technique well suited for the specification and verification of dependable systems possessing specific characteristics listed in 2.4.1.

The method used has its limitations which might be critical in other applications. For example, you might want to

- specify complex data structures
- use explicit and implicit specification styles in combination (see Chapter 3)
- simulate the specification (see Chapter 4)
- perform a risk analysis based on formal specifications
- describe hybrid systems
- use generic specifications allowing re-use for different projects (see Chapter 5)
- describe tightly coupled shared memory systems
- ...

This list is by no means considered as complete; it just gives an impression how many features might be desirable in the context of dependable systems development, but can never be united in a single method (and certainly never be implemented in a single tool). For these reasons, I do not believe that one universal method will emerge out of the (potentially non-terminating) competition of (formal) methods. Instead, different techniques will be applied to rather narrowly defined problem types, where they promise optimal efficiency. Indeed, I am convinced that a change of methods during the different development stages of one project will become quite natural in the future.

The advantage of combining different methods and associated tools leads to two important obligations:

- On the level of methodology it has to be ensured that objects described and results derived in the context of one method are consistently transformed into the framework of another method to be used in a further development step. Examples illustrating this obligation will be discussed in Chapter 3.
- On the tool level it has to be ensured that the transfer of development objects (specifications, code, test cases, proofs, ...) between different tools is not only "somehow" implemented, but consistent with the semantic properties of the associated methods. A specific problem is presented by the combination of semi-formal CASE tools with

formal specification and verification techniques, because reasoning in the formal framework requires precise semantics, and this is not *a priori* available for CASE tools in general.

Making use of the results developed in the ESPRIT project ProCoS (*Provably Correct Systems*) [10], these obligations will be analysed in detail in the UniForM project [88], proposed by the author in collaboration with ELPRO LET GmbH and the universities of Bremen and Oldenburg, and supported by the German Ministry *Bundesministerium für Bildung und Forschung BMBF*. Further applications of the framework as described in this chapter are planned in the context of the UniForM project (case study) and in the project [26] (large-scale development of a "real-world" system).

The investigation of formal semantics for CASE methods [20, 37, 113, 114] has been extensively investigated by the author in collaboration with de Roever, Hamer, Hörcher, Huizing and Petersohn [76, 79, 80, 81, 82, 84, 85]. Further notable efforts in this direction have been presented by other authors in [12, 8, 35, 36, 43, 45, 54]. However, the combination of CASE methods with formal methods is outside the scope of the topic of dependability. Therefore we will not discuss this issue further in the context of the *Habilitationsschrift*.

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].

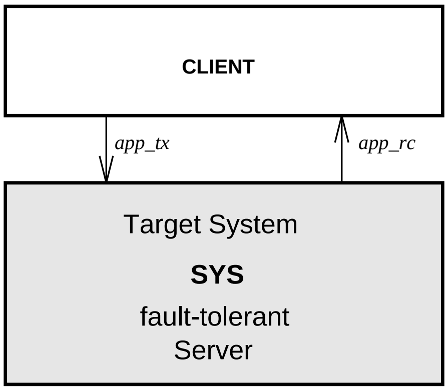

Figure 3.1: Client-server system.

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.

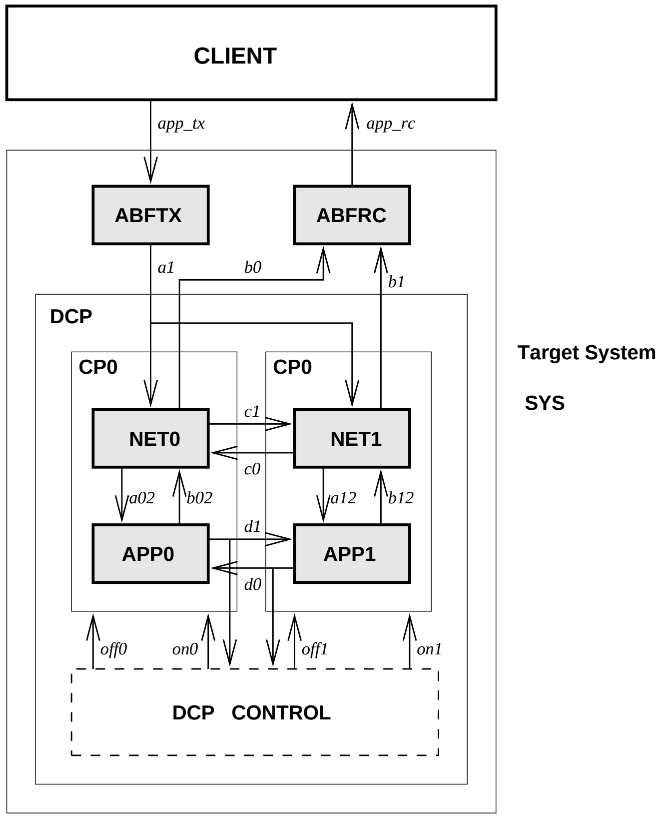

Figure 3.2: Full architecture of the fault-tolerant server system.

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

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

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:

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

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

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

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

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

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

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

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

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

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.

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

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

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.

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

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

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

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:

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

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

$$FH_{22} = x : \{\!|\ a_1\ |\!\} \to FH_{24} \,\square\, \mathit{on}_0 \to FH_2 \,\square\, x : \{\!|\ b_0, b_1\ |\!\} \to FH_{22}$$

$$FH_{23} = x : \{\!|\ a_1\ |\!\} \to FH_{25} \,\square\, \mathit{on}_1 \to FH_2 \,\square\, x : \{\!|\ b_0, b_1\ |\!\} \to FH_{23}$$

$$FH_{24} = x : \{\!|\ a_1\ |\!\} \to FH_{24} \,\square\, \mathit{on}_0 \to FH_{26} \,\square\, x : \{\!|\ b_0, b_1\ |\!\} \to FH_{22}$$

$$FH_{25} = x : \{\!|\ a_1\ |\!\} \to FH_{25} \,\square\, \mathit{on}_1 \to FH_{27} \,\square\, x : \{\!|\ b_0, b_1\ |\!\} \to FH_{23}$$

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

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

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.

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

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

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.

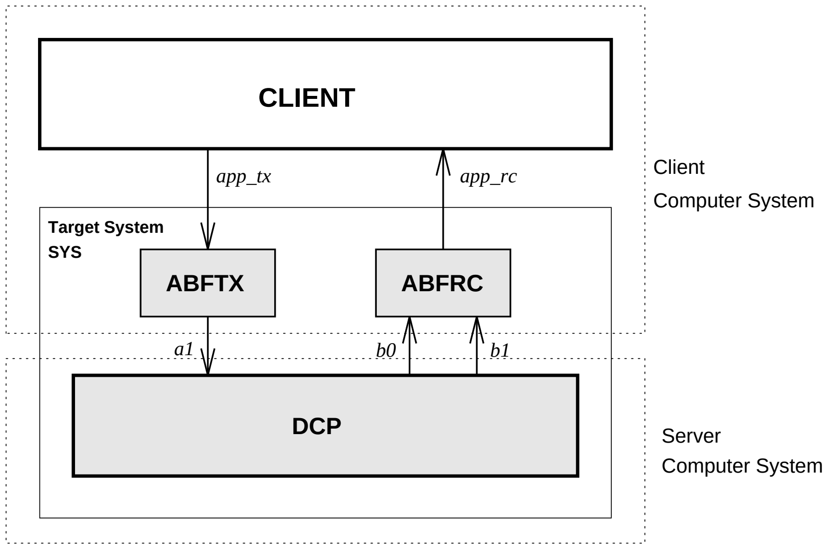

Figure 3.3: Server system architecture.

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

Component Specifications

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

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 $\parallel$ we know that [44, p. 128]

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

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

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

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 $S_P(s \upharpoonright \alpha(P), X)$ and $S_Q(s \upharpoonright \alpha(Q), Y)$ 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*

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

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 $\beta_n(u \upharpoonright (\alpha(P) - H)) = n \cdot (1 + \#(u \upharpoonright (\alpha(P) - H))) < \#(u \upharpoonright H)$. 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 $\#(\gamma(n)) \ge n$ 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 $(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC) \setminus \{\!|\ a_1, b_0, b_1\ |\!\}$. This specification is suitable to derive

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

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
$Div(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC) = \emptyset$ 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 $\parallel$ ([44, p. 128]) yields

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

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

Now let $u \in Div(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)$, such that $front(u) \notin Div(ABFTX \parallel ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \parallel ABFRC)$. 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

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

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 $(ABFTX \,\|\, ((DCP \setminus L_{DCP}) \wr \Delta_{DCP}) \,\|\, ABFRC)$ **sat** $NODIV(\{\!|\ a_1, b_0, b_1\ |\!\})$ holds.

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

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

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

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

As a consequence the estimate

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

and therefore

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

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

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

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

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

holds.

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

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

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

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

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

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

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 $\#(s \upharpoonright \{\!|\ app\_rc\ |\!\}) = \#(s \upharpoonright \{\!|\ app\_tx\ |\!\})$.

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

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

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

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

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

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

This completes the proof of deadlock freedom.

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

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

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

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

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)

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

on exiting the branch $state = 3$.

This time $post_3$ and the definition of ρ implies

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

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*

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

and

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

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

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

The operational semantics of assignment and deterministic sequential control structures is standard [5, p. 60]. $\mathbf{if}\ \square_{i=1}^{n} B_i \rightarrow S_i\ \mathbf{fi}$ 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* $\mathbf{do}\ \square_{i=1}^{n} B_i \rightarrow S_i\ \mathbf{od}$ 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

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

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

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

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

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

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*

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

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

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

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:

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

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 $\Box$, so that branches with guards evaluating to *false* do not contribute to the alternative construct. Therefore we calculate

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

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

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

we set

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

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

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

In correspondence with the P, the **do** $\ldots$ **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

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

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** $\ldots$ **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

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

In this expression S_L is an abbreviation for the program fragment

do $\ldots$ **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

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

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

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*

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

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

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

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

$\Sigma(P, \langle\ \rangle) =$

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

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

[interpretation of S_0 is the same in P and $\nu(P)$]

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

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

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

[semantic rules for assignment and sequential composition]

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

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

$$\Sigma(\nu(P), \langle\ \rangle)$$

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

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

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

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

[induction hypothesis]

$$\{\sigma, \sigma'' : \Sigma(\nu(P)) \mid < \nu(P), \sigma > \longrightarrow^* < S_L, \sigma'' > \wedge\ \sigma''(s_v) = s \wedge \sigma'' \models (\wedge_{(c,i) \in \Gamma_{IN} \cup \Gamma_{OUT}} \neg B_{c,i}) \bullet \pi_P(\sigma'')\} =$$

[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 $e = in.y_{in}$ and $in \in IN, y_{in} \in \alpha(in)$.

Then

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

[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]

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

[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 $s^\frown\langle e\rangle \in \mathit{Traces}(P)$ 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

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

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

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

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

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

$\xi(S)$ 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]

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

[Lemma 10]

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

[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:

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

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.

4. Safety Aspects: Test Automation for Reactive Systems

4.1 Overview

This chapter focuses on the problem of test automation for reactive systems. Its relationship to the *safety* aspect of dependability is twofold:

- The concepts described have been applied in practice for the test of safety properties in a tramway control system, and I expect that the main future applications will also be in the field of railway control, where safety properties play a dominant rôle among the dependability aspects.
- When comparing safety properties to *liveness*, especially *fairness* properties, the latter are less suitable for an analysis by means of testing: At least theoretically, a violation of fairness criteria can only be detected in *infinite* executions, while the objective of testing in practice is to find errors within a *finite* number of execution steps. As a consequence, the important properties of hard real-time systems and safety-critical systems will mostly be specified as safety properties.

For these reasons, the testing approach described in this chapter is mainly intended for the verification of dependable systems with emphasis on the safety aspect. Nevertheless you will notice that it is based on specifications where no explicit reference is made to *normal* and *exceptional behaviour* or *external* and *internal threats*, according to the notion of dependability introduced in Chapter 1. This has been justified in Chapter 2: At each stage of the development process for a dependable system it is possible to transform the collection of normal behaviour requirements, dependability requirements and fault hypotheses into "ordinary" design obligations, containing the dependability aspects as integral parts of the transformed specification. Such a transformed specification is the input document for the associated test procedure, and, as motivated above, it will contain mostly safety properties.

We describe the formal framework for the test system VVT-RT *(Verification, Validation and Test for Reactive Real-Time Systems)* developed by the author, which supports the automated test generation, test execution and test evaluation for reactive systems. VVT-RT constructs and evaluates tests based on formal CSP specifications [44], making use of their representation as labelled transition systems generated by the CSP model checker FDR [27]. A main objective of the present chapter is to provide a sound formal basis for the development and verification of high-quality test tools: Since, due to the high degree of automation offered by VVT-RT, human interaction becomes superfluous during critical phases of the test process, the trustworthiness of the test tool is an issue of great importance. Therefore it is intended to perform a verification suite for VVT-RT so that the tool can be certified for testing safety-critical systems by the responsible authorities in the fields of avionics and railway control systems. The present chapter represents the starting point of this verification suite, where the basic strategies for test generation and test evaluation used by the system are formally described and verified. VVT-RT has been designed to support automation of both untimed and real-time tests. The present chapter describes the underlying theory for the untimed case, which has also been published in [91, 93, 92]. Exploiting these results, the concepts and high-level algorithms used for the automation of real-time tests are described in a technical report which is currently prepared [89].

Section 4.2 describes the practical applications in industry that motivated the theoretical work presented here. In Section 4.3, we motivate and introduce the basic concepts of test automation in an informal way. In Section 4.4, some testing terminology used in industry is introduced together with explanations of the intuitive meaning of the corresponding terms. Several of these definitions will be formalised in the sections to follow. Section 4.5 provides a formal description and correctness proofs of the test generation and test evaluation mechanisms for *untimed* specifications, as implemented in the VVT-RT system. The main results are summarised and further research activities are described in Section 4.6.

4.2 Related Industrial Projects

Software Test for Avionic Software My practical work in the field of test automation started in 1990 with DST, when I was manager of a project carried out for Airbus Industries with the task to develop application and operating system software for the *A330/A340 Cabin Communication Data System (CIDS)* [1]. Since the hardware was developed in parallel with the software, we constructed a software simulation of the operating system layer and the system interfaces, to perform unit and integration testing for the application software in a UNIX environment. This *Soft Testbed* used a formal syntax for the generation of simulated system inputs and for recording the outputs stimulated by the application software.

While the tests performed in the soft testbed were much more effective and less time consuming than the system tests performed with the target system at a later stage, the test activities were still much too expensive. Moreover, experience showed that the test evaluation was not sufficiently trustworthy, since errors were sometimes overlooked during the manual analysis of the test output. These experiences motivated the test generation and automated test evaluation concepts based on the Z notation. In [85] a Z specification of a CIDS application function is described by Ute Hamer and myself, which served both for reliable software design and test design. Together with Hans-Martin Hörcher and Erich Mikk I investigated techniques for systematic test generation and automatic test evaluation against Z specifications [83]. This activity is still continued by Hörcher and Mikk in the context of a research project at Kiel University [41, 63].

Hardware-in-the-Loop Test for an Aircraft Engine Controller From 1992 to 1994 I managed a project at DST carried out for BMW-Rolls Royce. The task consisted in the development of a *hardware-in-the-loop test system* for the real-time test of the *Electronic Engine Control Computer* of the BR700 aircraft engine. In this system a powerful test language was used, allowing to define both digital and analog input data, stimulating the target system by means of hardware actors and receiving sensor outputs evaluated in the test system. A real-time simulation built of both software and hardware components implemented the environment behaviour as it was to be expected during real operation. A combination of manual and automatic analysis techniques supported the test evaluation process.

While the test system provided a considerable automation support, the test suite was based on structured, but informal requirements specifications. As a consequence, test procedures could not be generated in an automatic way. For the same reasons, the automatic test evaluation procedures could not be formally verified to be consistent with the specification, and it was not possible to calculate a formal measure for the test coverage achieved. These experiences motivated the investigation of test automation techniques based on formal methods suitable for the description of concurrent reactive systems[1].

Hardware-in-the-Loop Test for a Tramway Crossing Control System The work presented in this chapter was initiated in 1993, when I started a cooperation with ELPRO LET GmbH in the field of railway and tramway control systems (see also Section 2.2). Our joint activities focused on the improvement of *Verification, Validation and Test (VVT)* methods for such systems with the objective to increase the trustworthiness and improve the cost/benefit ratio of VVT activities by means of new methods and tools allowing to automate parts of the VVT process, thus far performed in a manual way. A first analysis of the different types of VVT activities to be performed for the development of safety critical systems resulted in the decision to tackle first the automation of hardware-in-the-loop tests[2]. For this type of test, the complete target system consisting of both software and hardware is analysed by a separate computer, the hardware-in-the-loop *test driver*. This driver connects to the target system using its operational interfaces and optionally some auxiliary channels to access internal system data. The driver stimulates the target system by sending data to its input channels and controls the system responses by monitoring the outputs. While exercising the tests on the target system, the test driver simulates the behaviour of the target environment, so that the hardware-in-the-loop test configuration behaves as close to the true operational conditions as possible. In this field the greatest benefit was expected, because

- hardware-in-the-loop tests can be used to prepare the final acceptance test which is the most critical VVT activity with respect to the success of the project,
- the most severe errors were expected in the interplay between software and hardware, so that they could not be detected by means of isolated unit tests or integration tests performed in a simulation environment[3].

These considerations motivated the development of the VVT-RT system together with its underlying theory described in this chapter. The first application which has recently been

[1]However, I am not convinced that a fully automated test against a formal specification would have been possible in the BMW-Rolls Royce application: The formal specification language would have been a language suitable for the description of *hybrid systems*, and the possibilities to simulate such specifications are still restricted. Moreover, the relationship between formal hybrid system descriptions and testing still remain to be investigated.

[2]Further improvements of the development process and the VVT process are currently investigated in the UniForM project [88].

[3]This assessment of error criticality is consistent with with the estimates described bei Wegener *et al.* in [115]: The authors state that from their experience with the evaluation of total testing expenses in projects performed at Daimler-Benz AG the main costs are caused by system testing in combination with the examination of software-hardware interfaces (38%). 34% of the total expenses are spent on unit testing, and 27% for integration testing.

completed was the hardware-in-the-loop test of a tramway crossing control computer developed by ELPRO LET GmbH and based on PLC hardware [25, 86, 87, 90, 92]. The task of this system is to detect trams approaching the crossing by means of electro-mechanical sensors and switch traffic lights and signals according to an algorithm assigning priority to trams, at the same time guaranteeing time intervals when pedestrians and vehicles may pass the crossing. Dependability issues consist in a combination of safety and availability requirements: To provide safety, the system must never allow signals and traffic lights to be in state "green" at the same time and observe time intervals where the crossing is completely blocked before granting access to trams or road vehicles. In case of a system failure, a *stable* safe state is assumed by switching off the signals and turning the yellow lights into flash mode, which indicates that every vehicle passing the crossing has to do so on its own responsibility. To increase the reliability, redundant hardware is used in signals, traffic lights and their electronic interfaces. The software of the control system detects failures concerning the current and voltage of these peripherals. It implements a switching concept ensuring that in case of a component failure either spare components can be activated or the system performs a transition into the stable safe state.

The test was executed on a hardware platform developed by ELPRO, consisting of a PC implementing a test driver and a PLC system acting as interface between PC and the target system PLC. The formal specification of the target system, the test generation and evaluation were performed by myself. The test results were considered as successful [90] for the following reasons:

- Two errors were found by means of VVT-RT, after the target system had been thoroughly tested in a manual way and passed the official acceptance test performed by the certification authorities.
- The number of tests performed and documented automatically by VVT-RT resulted in about 3600 pages of test documentation[4] which would otherwise have to be manually produced to document the same degree of test coverage without tool support.
- Though the test procedure using VVT-RT was performed for the first time, the total costs of the automated tests were less than 30% of the costs estimated for an equivalent manual test.

4.3 Test Automation for Reactive Systems – Motivation and Basic Concepts

4.3.1 Motivation

Design, execution and evaluation of trustworthy tests for safety-critical systems require considerable effort and skill and consume a large part of today's development costs for software-based systems. It has to be expected that with conventional techniques, the test coverage to be required for these systems in the near future will become technically unmanageable and lead to unacceptable costs. This hypothesis is supported by the growing complexity of applications and the increasingly strict requirements of certification authorities with respect

[4]This corresponds to 98% branch coverage of the transition graph representing the specification.

to the verification of safety issues. For these reasons methods and tools helping to automate the test process gather wide interest both in industry and research communities. *"Serious"* testing – not just playing around with the system in an intuitive unsystematic way – always has to be based on some kind of specification describing the desired system behaviour at least for the situations covered by the test cases under consideration. As a consequence, the problem of test automation is connected to formal methods in a natural way, because the computer-based design and evaluation of tests is only possible on the basis of formal specifications with a well-defined semantics.

Whereas it may be argued that the test of software code may become superfluous in the future, because programs may be derived from formal specifications by means of stepwise refinement accompanied by formal verification or by means of formally verified transformational techniques, I am convinced that testing will always remain mandatory for the analysis of behaviour *on system level*:

- Formal development and verification procedures will never cover the full range of software and hardware components, at least for medium-sized and large systems.
- As Brinksma points out in [11] in the context of *conformance testing*, manufacturers will not always disclose the implementation details of their products if external groups perform the product evaluation. As a consequence, (black-box) testing will be the only means to investigate the correctness properties of such a system.
- While the correctness properties of formally verified software will not "wear out" during system operation[5], hardware components initially functioning correctly may fail after a period of operation which is only statistically predictable. As a consequence, safety-critical systems have to be analysed in regular intervals using tests designed to detect "local" failures before they can impair the system functionality required by the user. For example, critical computer components in airborne systems have to perform continuous testing in all flight phases using their *Built-in Test Equipment (BITE)* [22].
- A psychological, but nonetheless even more important reason is that customers prefer to see a system in operation before paying the bill. An acceptance test with the real system will never be replaced by a document review of all the formal verification activities performed during system development.

Just as it is impossible to build theorem provers for the fully mechanised proof of arbitrary assertions, the general problem of testing against arbitrary types of specifications cannot be solved in a fully automated way. The situation is much more encouraging, however, if we specialise on well-defined restricted classes of systems and test objectives. This strategy is pursued in our approach, where we will focus on the *hardware-in-the-loop test* of *reactive systems*.

The idea to apply the theoretical results about testing in process algebras to practical problems was first presented by Brinksma, with the objective to automate testing against LOTOS specifications. His concept has been applied for the automation of *OSI conformance tests*; see [11] for an overview. Today, testing against different types of formal specifications has gained wide interest both for engineers responsible for the quality assurance of safety-critical

[5] As long as the requirements remain constant over time!

systems and in the formal methods community: To name a few examples, Gaudel [30] investigates testing against *algebraic specifications*, Hörcher and Mikk in collaboration with the author [83, 41, 41, 63] focus on the automatic test evaluation against *Z specifications* and Müllerburg [68] describes test automation in the field of *synchronous languages*.

The objective of our contribution presented in this chapter and in [89] is not to present a new testing theory, but to create "implementable instantiations" of well known theoretical results with the objective of dependable tool construction. Our approach is based on the untimed CSP process algebra and uses Hennessy's testing methodology [39] as a starting point. In [89], we expand these results into testing against timed CSP specifications, with an emphasis on the investigation of timed safety properties, based on the semantics presented by Davies [18]. The full specification will be separated into timed and untimed components, where the interpretation of the latter is consistent with Hoare's untimed CSP [44], but must representable as finitely generated transition systems.

4.3.2 Formal Methods for Tool Qualification

While the availability of a test automation tool offers the possibility to increase the number of test executions by a substantial amount, it has to be taken into account that this will also lead to situations where at most a small subset of the test definitions and results may be checked in a manual way. Indeed, the full scale of advantages offered by the tool can only be exploited if users can trust the correct operation of the tool completely, without manual inspection of the test procedures and results. This problem has been recognised by several authorities responsible for the certification of safety-critical systems. For example, the development standard [22] requires all tools to be verified and certified according to strict evaluation procedures, as soon as they automate any development or verification activity in a way making human interaction superfluous. Due to these strict evaluation requirements, the number of tools certified for the automated development or verification of safety-critical systems is very small. Furthermore, it is often criticised that the tools will be out-dated before the evaluation and certification suite has been completed. This situation may be considerably simplified, if a tool is developed from the beginning with the intention to allow evaluation and certification. The starting point of such a development process should always be a precise specification of the application range intended for the tool, followed by a description and verification of the techniques used to implement the specified services. Such a starting point is provided by this chapter: We present formal specifications for the notion of *trustworthy* test tools and mathematical proofs for the correctness of *test drivers* performing automatic test generation and test evaluation.

4.3.3 Logical Building Blocks of a Test Automation System

To tackle the full task of tool qualification in a well-organised way, let us look at the logical building blocks of a complete test automation system (Figure 4.1)[6].

The Test Generator is responsible for the creation of test cases from specifications. The generator is called trustworthy, if for each possible implementation error violating the specified

[6]Figure 4.1 is not intended to suggest a system architecture for a test system, it only represents logical components that should be present.

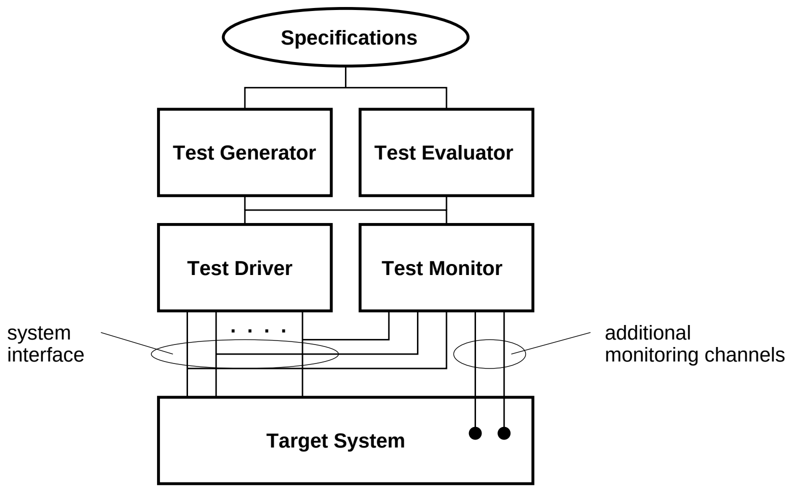

Figure 4.1: Logical building blocks of a test automation system.

requirements a test case will be created which is capable of detecting this error. Observe that in the context of reactive systems, a test case is not simply a sequence of inputs and expected outputs: Since the target system may behave nondeterministically on the interface level, the same sequence of inputs may stimulate different responses of the target system, or may even be refused during repeated executions. Therefore we rather define a test case as a description of *execution rules*, specifying the full set of possible sequences of input and output events, together with *real-time assertions* which should be met when exercising the test case on the target system.

The Test Driver interprets the test cases provided by the test generator and controls their execution by writing data on input channels of the target system and collecting system outputs. In general, the target system is only required to behave according to the specification, as long as the operational environment also behaves correctly. A test driver is therefore called trustworthy, if it exactly simulates the behaviour of the "real" environment during test execution.

The Test Evaluator verifies the observed test execution against a specification and decides whether the execution was correct. The specification document used for test evaluation, say, $SPEC_2$ is not necessarily the same as the specification $SPEC_1$ used for test generation: For example, $SPEC_1$ may be an explicit CSP specification, because the possibility to interpret explicit CSP processes is suitable for test generation. For test evaluation, it may be preferable to use trace assertions in $SPEC_2$. Moreover, it will not always be the case that every behavioural property reflected by $SPEC_1$ will also be checked by $SPEC_2$. Conversely, it may be the case that $SPEC_1$ is an incomplete description of the required behaviour, which is just suitable for the generation of certain test cases, while $SPEC_2$ is a stronger specification, valid for the evaluation of other classes of test cases as well. For the test evaluator trustworthiness means that, given the results of a test execution, it will be detect every violation of the specified requirements.

The Test Monitor observes each test execution in order to decide whether

- a specific test case has been performed for all relevant executions that are possible for this case, when exercised on the target system,

- the full set of test cases executed so far suffices for the required level of test coverage.

In many applications, these tasks of a test monitor cannot be performed based on the black-box observations of the system interface alone. Instead, additional channels must be created, providing internal state information about the target system for the monitor.

4.4 Testing Terminology

In this section some testing terminology, as presently used in industry is introduced. The definitions have been taken from the international standard DO-178 B [22] for the development of software in airborne systems. *Verbatim* quotations from [22] are displayed as *emphasised* text.

The term *Validation* will denote the process of *determining that the requirements are the right requirements and that they are complete.* Since the system requirements specification is the "initial" document describing the system behaviour desired, no other document exists which could serve as a reference to *prove* that the specification is complete and really describes the customer's demands. Therefore validation can never be performed by means of formal proofs alone, but must also rely on informal techniques like simulation or – if the customer and the supplier will risk an *a posteriori* validation – system testing. *Verification* denotes an evaluation of development products (specification documents, design documents, code etc.) with the objective to ensure their consistency with respect to the reference documents applicable for the products. So in contrast to validation, the verification process can rely on other documents specifying (at least in theory) completely how the product should look like. If the reference documents are written using a description language with a precisely defined semantics and verification is performed by means of mathematical reasoning, the term *Formal Verification* will be used[7].

In our context *Testing* means execution of implemented system components providing specific data at their (input) interfaces, while monitoring the component behaviour. The objective of testing is to *verify that the component satisfies the specified requirements for the set of input data applied.* To avoid confusion we will use the term *Simulation* for the *symbolic execution* of a specification with specific (abstract) input data, though this is nothing else than testing on an abstract level.

A *Test Case* is a *set of test inputs, execution conditions, and expected results developed for a particular objective.* A test case does not necessarily define the complete set of input and output data to be used in an explicit way. Instead, the test case can consist of a specification describing how to construct the explicit *test data*, starting with initial inputs and deriving consecutive data values and expected results during the test execution. The objective of *expected results* is to provide an unambiguous characterisation of correct system behaviour, as it may be observed during an execution of the test case. Since "correct" behaviour is defined by the specification document, the expected results – if developed in the correct way – should be consistent with the specification.

[7] Observe that this definition of *verification* differs from that normally used in the formal methods community, where it is always supposed to be the *formal* verification process. In the context of this chapter, verification subsumes all formal *and* informal techniques applicable for *a posteriori* insurance of product system correctness, i. e., reviews, audits, walk throughs, test, formal verification etc.

A *Test procedure* is a *detailed instruction for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.*

We will use the term *requirements-driven testing* if the specification serves as the input document to construct the associated test cases. If the test cases are derived from the implementation, e. g. from the software code, the term *implementation-driven testing* will be used. Note that implementation-driven testing also requires a specification document, because otherwise it cannot be decided whether the results are correct.

Tests may be exercised on the complete system as well as on isolated system components. In practice, strategies for a combination of tests on system level and tests on component level are developed, in order to cope with the overall complexity of the test process. The term *Unit Testing* will be used to denote the test of one isolated system component, if no smaller unit of this component will be tested separately. Depending on the granularity used for the test of a specific system, a unit may denote a block, a function, a thread, a task or any other coherent software unit. *Integration Testing* applies a strategy of successively adding new components to those already tested and then performing a test of the resulting configuration. *System Testing* denotes the test of the complete target system. For system testing, the environment used for the tests will be the "real" operational context or at least a simulation implemented according to the hypotheses about the operational environment behaviour at the interface level of the target system.

Tests influencing and monitoring only the interface behaviour of the component to be tested will be called *Black-Box Tests*. If internal states or internal execution sequences of the component are also monitored or even influenced, this will be denoted as *White-Box Tests*.

If a component is not tested in the operational environment of the system but within an "artificial" context, the software and hardware implementing this context will be called a *Test Driver*.

As pointed out for example by Hennessy [39], test cases for reactive systems cannot be simply described by means of a pre-state, test inputs and the expected state "after program execution". Since a reactive system interacts continuously with its environment, test cases also have to verify the correct causal relationships between inputs and outputs, the readiness to engage into specific actions, real-time requirements etc. Therefore in the context of reactive systems a *(Black-Box) Test Execution* denotes a *Trace* observable at the interface between the test driver and the component under consideration. In our context a trace is a finite sequence of *Events* (t, c, v). Here c denotes an input- or output interface channel observable by the test environment. We do not impose any restrictions about one-to-one or multi-way communication. Value v is passed over channel c. *Time stamp* t records the point in time, when the communication took place.

A *test case for reactive systems* consists of the following information:

1. *input data and causal execution conditions*: This is a specification referring to the channels of the component to be tested. It describes initial inputs and consecutive inputs depending on the reactions of the target system. The specification may be defined by means of a predicate or by an explicit process characterising the behaviour required of the test driver during test execution.

2. *timed execution conditions*: a specification defining time-dependent conditions when

to stimulate a certain input

3. *(timed) expected results*: a specification characterising the traces which are regarded as correct executions of the test case. In the case of deterministic reactive systems without timing requirements, this specification may explicitly define the sequence of inputs and outputs which is required. However, as soon as timing requirements exist or tolerances for data values are acceptable or the system reaction to a sequence of inputs may be nondeterministic, correct system behaviour can only be specified by more general assertions or processes.

In specific situations, where the systems to be verified can only assume a very limited number of different states and use only restricted ranges for variable values, we may test every possible execution performable by the system. If this complete number of tests is reached, the process will be called *exhaustive testing*.

4.5 Trustworthy Testing – Untimed Case

4.5.1 Motivation and Conceptual Background

The objective of this section is to investigate methods for the automated tests of systems against specifications expressible in *untimed CSP* as defined by Hoare [44]. The basic idea for the automation process is:

- Automated test generation is performed by mechanised analysis of the specification, which results in a choice of traces and acceptance sets to be exercised as test cases on the target system.
- Automated test evaluation is performed by observing traces and readiness of the target system and checking mechanically, whether this behaviour is correct according to the specification.

Obviously, these tasks are fundamentally connected to the problem of mechanised *simulation* of the specification. How to find an implementable solution for this problem, is indicated by the following well-known theorem (see, e. g., [39, p. 94]):

Lemma 12 (Normal Form Theorem) *Let P be a CSP process, interpreted in the failures-divergence model.*

1. *If $\langle\,\rangle \notin Div(P)$, then*

$$P = \sqcap_{R:Ref(P)}(x : (P^0 \setminus R) \rightarrow P/\langle x \rangle)$$

2. *If $Div(P) = \emptyset$, then $P/s = P(s)$ with*

$$P(s) = \sqcap_{R:Ref(P/s)}(x : ([P/s]^0 \setminus R) \rightarrow P(s^\frown\langle x \rangle))$$

3. *For arbitrary P, $P \sqsubseteq_{FD} P(\langle\,\rangle)$ holds.*

□

Here the notation $[P/s]^0 = \{e : \alpha(P) \mid (\exists\, u \in \mathit{Traces}(P/s) \bullet \mathit{head}(u) = e)\}$ has been used and P/s denotes the process P after having performed a trace s.

Interpreting this result shows how *trustworthy simulators* – i. e., simulators really implementing what is expressed by the specification – could be constructed: At each step P/s of a symbolic execution, choose a valid refusal set R at random and engage into any one of the remaining events $e \in [P/s]^0 \setminus R$. After some e has been chosen, continue in state $P/s^\frown\langle e\rangle$. Given an implementation of a simulator, the problem of test generation can be related to the task of finding executions performable by the simulator. Test evaluation can be performed by determining whether an execution of the real system is also a possible execution of the simulator.

With these general ideas in mind, the initial problem to solve is how to retrieve the semantic representation – i. e., the failures (traces plus refusals) and divergences – of a specification written in CSP syntax. This has been solved by Formal Systems Ltd and implemented in the FDR system [27], for the subset of CSP specifications satisfying

- The complete CSP specification only uses a finite alphabet. As a consequence, each channel admits only a finite range of values.
- Each sequential process which is part of the full specification may be modelled by means of a finite number of internal states.
- The CSP syntax is restricted by a separation of operators into two levels: The *lower-level process language* describes isolated communicating sequential processes by means of the operators $\rightarrow$, $\sqcap$, $\Box$, $;$, $X = F(X)$. The *composite process language* uses the operators $\|$, $|||$, $^\frown$, $\setminus$, f^* to construct full systems out of lower-level processes.

Under these conditions the CSP specification may be represented as a *Labelled Transition System (LTS)* [64] which may be encoded by means of a *Transition Graph* with only a finite number of nodes and edges. Basically, the nodes of this directed graph are constructed from Hennessy's *Acceptance Tree* representation [39] by identifying semantically equivalent nodes of the tree in a single node of the transition graph. The edges of the graph are labelled with events, and the edges leaving one node carry distinct labels. Therefore, since the alphabet is finite, the number of leaving edges is also finite. A distinguished node represents the equivalence class of the initial state P of the process. A directed walk through the graph, starting in the initial state and labelled by the sequence of events $\langle e_1, \ldots, e_n\rangle$ represents the trace $s = \langle e_1, \ldots, e_n\rangle$ which may be performed by P, and the uniquely determined node reached by the walk s represents the equivalence class of process state P/s. The labels of the edges leaving P/s define the set $[P/s]^0$ of events that may possibly occur in state P/s. In the *failures model* of CSP a state P/s is characterised by $[P/s]^0$, together with the set of associated refusals $\mathit{Ref}(P/s)$. Since the alphabet is finite and the collection of refusals is subset-closed, $\mathit{Ref}(P/s)$ is uniquely determined by the set $\mathit{refMax}(P/s)$ of *maximum refusals*: $\mathit{Ref}(P/s) = \{X : \mathbb{P}\,\alpha(P) \mid (\exists\, R : \mathit{refMax}(P/s) \bullet X \subseteq R)\}$.

The LTS representation of P by means of a *finite* transition graph is possible, because any CSP process constructed by communicating finite-state sequential processes using the composite process language is also finite-state. Therefore, though P may be a non-terminating process, only a finite number of processes P/s and P/u will be distinct, and an infinite acceptance tree representation of P may be encoded as a finite graph by identifying the nodes P/s and P/u, if the corresponding processes are identical in the failures model. So, if two directed walks s and u lead to the same node in the transition graph, this means that $P/s = P/u$ holds in the failures model.

Encoded in this way, the transition graph therefore allows to recover the complete failures set of a process P, consisting of all pairs (s, X), where s is a trace of P and X a refusal of P/s. Together with the alphabet, this is the representation of P in the (denotational) failures semantics. The possibility of *divergence* for a process $Q = P \setminus H$ may be detected by finding a cycle in the transition graph of P using only events contained in H.

Formal Systems' FDR tool compiles CSP specifications into their corresponding transition graph. As a consequence, any evaluation of the specification semantics may be implemented by means of programs analysing this finite graph.

The problem of automatic test evaluation is now rather trivial: A test execution results in a trace performed by the implementation. Evaluating the transition graph, it may be verified whether this execution is correct according to the specification. The problem of "trustworthy" test generation is much more complex: Theoretically, the transition graph defines completely the acceptable behaviour of the implementation and – by taking traces and refusals *not* occurring in the graph – the incorrect behaviour of an implementation. For non-terminating systems, this involves an infinite variety of possible executions. Therefore the problem how to find "relevant" test cases and how to decide whether "a sufficient amount" of test executions has been exercised on the target system has to be carefully investigated.

4.5.2 CSP, Refinement and the Relation to Testing Methodology

In this section some well-known properties about the relation between testing and refinement will be summarised and re-phrased for the context of this chapter. Intuitively speaking, a testing procedure should be called "trustworthy", if the associated collection of test cases "converges" to some kind of "correctness proof" with each additional successful test execution. In order to define these informal notions in a precise way, at first we have to decide about an appropriate definition of "correctness".

Correctness Definitions for Implementations

Since we are concerned with the test of an implementation *IMP* against its specification *SPEC*, it is natural to choose a relation between *SPEC* and *IMP* to define the correctness of the implementation. We assume that both *SPEC* and *IMP* may be expressed as untimed CSP processes. In this case the intuitive understanding of implementation correctness in the context of safety-critical systems may be formally expressed by means of different types of CSP *refinement relations*. To this end, it will be required that $\alpha(SPEC) = \alpha(IMP)$, so specification and implementation are both restricted to the set of events that are "relevant" for the decision of implementation correctness. This does not mean, that specification and implementation have to be completely constructed on the basis of the same set of events: In general $SPEC = X \setminus (\alpha(X) - I)$ and $IMP = Y \setminus (\alpha(Y) - I)$, where X and Y contain specification and implementation details using different alphabets. However, we assume that the correctness of *IMP* with respect to *SPEC* may determined by observing I-events only. This will be investigated in the context of tests for *reactive systems* in Section 4.5.3.

The relation between intuitive correctness notions and their formal CSP equivalents can be described as follows:

1. **Safety:** The implementation should only perform executions pre-planned in the specification. This is can be expressed by demanding that the trace space of the implementation should be contained in that of the specification, i. e.

$$Traces(IMP) \subseteq Traces(SPEC)$$

2. **Requirements Coverage:** After having performed an initial execution s which is possible both for specification and implementation, the implementation should never refuse a service which is not refused by specification. Formally speaking,

$$\forall\, s : Traces(SPEC) \cap Traces(IMP) \bullet Ref(IMP/s) \subseteq Ref(SPEC/s)$$

   Since $\langle\,\rangle \in Traces(SPEC) \cap Traces(IMP)$, this implies that a trace which can never be refused by $SPEC$ will also be performed by IMP. However, if $Traces(SPEC)$ contains $s ^\frown \langle e \rangle$ but may refuse e after s, it is not guaranteed that the implementation will be able to perform $s ^\frown \langle e \rangle$: Sufficient requirements coverage allows to refuse a service in the implementation *completely* if it *may* be refused according to the specification.

3. **Non-Divergence:** Apart from blocking a service, non-availability may be caused by performing an infinite sequence of internal events invisible to the outside world, or by showing completely unpredictable system behaviour. This is denoted by *divergence*, and the implementation should diverge – if at all – only after traces where the specification also diverges.

$$Div(IMP) \subseteq Div(SPEC)$$

4. **Robustness:** If it is required that *every* trace pre-planned in the specification should also be performable in the implementation, we need

$$Traces(SPEC) \subseteq Traces(IMP)$$

   This definition of robustness, introduced in [11], has not received much attention in the literature about CSP refinement, though it is a common requirement in practical applications: For example, robustness covers the situation where the specification process contains nondeterminism for exception handling. Failures refinement only requires that every guaranteed behaviour of the specification process will also be performed by the implementation. Robustness additionally requires that nondeterministic exceptional behaviours described by the specification process are also covered by the implementation.

Example 4.1 Let $SPEC = a \rightarrow b \rightarrow STOP \sqcap c \rightarrow d \rightarrow STOP$ Then $IMP_1 = a \rightarrow STOP$ is a safe implementation of $SPEC$, but does not provide requirements coverage. $IMP_2 = a \rightarrow b \rightarrow STOP \;\square\; e \rightarrow STOP$ provides requirements coverage but is neither robust nor safe. $IMP_3 = a \rightarrow b \rightarrow STOP \;\square\; c \rightarrow d \rightarrow STOP$ is safe and provides requirements coverage and

robustness.
□

These correctness properties will now be related to the notions of *trace (T)-refinement* $\sqsubseteq_T$, *failures (F)-refinement* $\sqsubseteq_F$, *failures-divergence (FD)-refinement* $\sqsubseteq_{FD}$ and *failures-divergence-robustness (FDR)-refinement* $\sqsubseteq_{FDR}$. Recall that (see [44, 27, 69])

1. $SPEC \sqsubseteq_T IMP \equiv Traces(IMP) \subseteq Traces(SPEC)$
2. $SPEC \sqsubseteq_F IMP \equiv Fail(IMP) \subseteq Fail(SPEC)$
3. $SPEC \sqsubseteq_{FD} IMP \equiv SPEC \sqsubseteq_F IMP \wedge Div(IMP) \subseteq Div(SPEC)$
4. $SPEC \sqsubseteq_{FDR} IMP \equiv SPEC \sqsubseteq_{FD} IMP \wedge IMP \sqsubseteq_T SPEC$

The following table describes the correspondence between the correctness properties introduced above and these refinement notions ('•' means that the property is implied by the refinement relation):

Property	$\sqsubseteq_T$	$\sqsubseteq_F$	$\sqsubseteq_{FD}$	$\sqsubseteq_{FDR}$
safety	•	•	•	•
requirements coverage		•	•	•
non-divergence			•	•
robustness				•

Apart from this obvious correspondence to the intuitive correctness notions, there is another reason to focus on these refinement concepts when investigating methods for test automation: As described in the next section, Hennessy's testing methodology introduces types of test cases uncovering exactly those implementation failures which violate one of the above refinement relations. As a consequence, the compositional proof theory associated with the refinement concepts may be consistently used in parallel with the testing process: Critical components may be formally verified to fulfill the refinement relation desired. At the same time tests may be performed for less critical components, and these tests may be regarded as "approximations" of refinement proofs or "incomplete formal verifications". In specific cases to be investigated during the next sections, it is even possible to perform *exhaustive testing* with a finite number of test executions, so that the test activities represent a valid substitute for the corresponding refinement proof.

In some applications one might prefer to specify the correct behaviour of an implementation by means of an assertion $IMP \overset{!}{\textbf{sat}} S(h, R)$ where $S(h, R)$ is a predicate with trace h and refusal R as free variables. However, this concept of correctness is – at least theoretically – covered by our definition: If predicate $S(h, R)$ can be satisfied by a CSP process at all, then there exists a most nondeterministic process $SPEC$ such that $SPEC$ **sat** $S(h, R)$ [100]. As a consequence, verifying whether IMP satisfies the assertion can be viewed as the proof obligation $SPEC \sqsubseteq_{FD} IMP$. We prefer to work with the $\sqsubseteq$ correctness concept only, because in order to allow automatic generation of test cases in the VVT-RT system, $SPEC$ will have to be represented as an explicit CSP process.

Hennessy's Testing Methodology

As shown by Hennessy and de Nicola [39, 69], trace *(T)*-, failures *(F)*-, failures-divergence *(FD)*- and failures-divergence-robustness *(FDR)*-refinement are closely related to testing. The results summarised below hold in an even wider context as it is needed for the purpose of this chapter, see [40]. Extending Hennessy's original concepts, several terms and definitions to be used for the classification of tests in the subsequent sections are introduced. These definitions are not essential if the testing methodology is applied only for the characterisation of process algebra semantics, but they are useful for the practical applications of the testing methodology in the context of reactive systems.

Hennessy introduced *experimenters* as processes U with $\alpha(SPEC) = \alpha(IMP) \subset \alpha(U)$ and a specific event $w \in \alpha(U) - \alpha(SPEC)$ denoting successful execution of the experiment which consists of U running in parallel with a process possessing alphabet $\alpha(SPEC)$. Experimenters coincide with our notion of *Test Cases*, so we will only use the latter term. As pointed out in Section 4.4 an execution of the test case U for the test of some system P is a trace $s \in \mathit{Traces}(P \parallel U)$. The execution is successful if $s ^\frown \langle w \rangle \in \mathit{Traces}(P \parallel U)$. Depending on the specification of U and P, three cases may be distinguished with respect to the outcome of test executions:

1. There exists at least one successful execution of $(P \parallel U)$.
2. Every execution of $(P \parallel U)$ must lead to success.
3. No execution of $(P \parallel U)$ is successful.

This observation leads to the following formal definition:

Definition 1 *For a process P and an associated test case U we say*

1. $P \ \underline{may}\ U \equiv (\exists\, s : \mathit{Traces}(P) \bullet s ^\frown \langle w \rangle \in \mathit{Traces}(P \parallel U))$
2. $P \ \underline{must}\ U \equiv (\exists\, Q \bullet (P \parallel U) \setminus (\alpha(U) - \{w\}) = w \to Q)$

□

The definition of P <u>*must*</u> U requires some explanation: If every execution of $(P \parallel U)$ leads to success, this is equivalent to the following conditions:

- $(P \parallel U)$ may never block before having produced a success indication w.
- $(P \parallel U)$ may never engage into an unbounded sequence of events not containing w.

After success has been indicated, we do not care about the further behaviour of $(P \parallel U)$. This situation is precisely reflected by $(P \parallel U) \setminus (\alpha(U) - \{w\}) = w \to Q$, where "=" denotes equivalence in the failures-divergence model. This algebraic definition is equivalent to Hennessy's definition [40] of <u>*must*</u>-tests, requiring that every *computation* of $(P \parallel U)$ should indicate success.

Note that we cannot demand that test cases should indicate *failure* in addition to success, because the failure may materialise as a situation where the test execution is blocked or diverges, so that it can only be "communicated" by *refusing to indicate success*. This situation is simplified when studying real-time testing for safety-critical systems: Here expected events should always occur within a bounded amount of time, so deadlock or livelock can be detected by means of timeouts.

If a test execution refuses to indicate success, this information will only be of practical value, if it can be determined what actually went wrong. The correctness notions defined by *T,F,FD,FDR*-refinement allow a precise classification of the failures in an implementation *IMP* when compared with its specification *SPEC*. This classification leads to the following definition:

Definition 2 *Let U be a test case.*

1. *U detects a* safety failure s *iff* $(\forall\, P \bullet P\,\underline{must}\,U \Rightarrow s \notin \mathit{Traces}(P))$
2. *U detects a* requirements coverage failure (s, A) *iff* $(\forall\, P \bullet P\,\underline{must}\,U \Rightarrow (s, A) \notin \mathit{Fail}(P))$
3. *U detects a* divergence failure s *iff* $(\forall\, P \bullet P\,\underline{must}\,U \Rightarrow s \notin \mathit{Div}(P))$
4. *U detects a* robustness failure s *iff* $(\forall\, P \bullet P\,\underline{may}\,U \Rightarrow s \in \mathit{Traces}(P))$

□

For practical reasons it is useful to restrict the class of admissible test cases: If the execution is successful we want to be sure to be informed about this "at once". Moreover, the indication of success – if possible – should not allow an unbounded trace of other events to be executed before the "success situation" arises. Since test cases will be executed by test drivers which must know when a test execution has been successfully completed, we require a well-defined termination event after having signalled success, and it is not necessary to indicate success more than once. Finally, we restrict ourselves to test cases which are successful as $\underline{must}$-tests for at least one process. This leads to the following definition:

Definition 3 *An* admissible test case *for the test against SPEC is a CSP process U satisfying*

1. $\alpha(U) = \alpha(\mathit{SPEC}) \cup \{w, \checkmark\}$, $w \notin \alpha(\mathit{SPEC})$
2. U **sat** $S_U(s, R)$ *with*

$$S_U(s, R) \equiv \\ w \in [U/s]^0 \Rightarrow w \notin R \wedge \#s \leq n \wedge \neg\,(\langle w \rangle \textbf{ in } s) \wedge U/s^\frown\langle w \rangle = \mathit{SKIP}$$

   where $n \in \mathbb{N}$ *is a constant not depending on* s *or* R.

3. *There exists a process P such that* $P\,\underline{must}\,U$.

□

The following examples illustrate the intuition standing behind the above definition by presenting test cases that are *not* admissible.

Example 4.2 The test case $U = a \rightarrow \mathit{SKIP} \,\square\, b \rightarrow (w \rightarrow \mathit{SKIP} \sqcap U)$ would not be admissible in the sense of Definition 3, because it is uncertain whether success will be indicated after

event b.
□

Example 4.3 The test case $U = a \to w \to SKIP \sqcap STOP$ would not be admissible in the sense of Definition 3, because no process can satisfy U as a $\underline{must}$-test.
□

Example 4.4 The test case

$$U = \sqcap_{n:\mathbb{N}} U(n)$$
$$U(n) = (n > 0)\&a \to U(n-1) \;\Box\; (n = 0)\&w \to SKIP$$

would be well-defined in the infinite traces model of Roscoe and Barret [99], and $P \; \underline{must} \; U$ holds for process $P = a \to P$. Moreover, if success w is possible after U/s it will never be refused. However, U would not be admissible in the sense of Definition 3, because no global upper bound exists after which every execution of $(P \parallel U)$ would show success.
□

Hennessy defines four types of test cases which are essential for the verification of T,F,FD,FDR-refinement properties:

Definition 4 *The class of* Hennessy Test Cases *is defined by the following collection of admissible test cases:*

1. *For* $s \in \alpha(SPEC)^*, a \in \alpha(SPEC)$, *define* Safety Tests $U_S(s, a)$ *by*

$$\begin{aligned} U_S(s,a) = \;&\mathbf{if}\; s = \langle\,\rangle \\ &\mathbf{then}\; (w \to SKIP \;\Box\; a \to SKIP) \\ &\mathbf{else}\; (w \to SKIP \;\Box\; (head(s) \to U_S(tail(s), a)) \end{aligned}$$

2. *For* $s \in \alpha(SPEC)^*, A \subseteq \alpha(SPEC)$, *define* Requirements Coverage Tests $U_C(s, A)$ *by*

$$\begin{aligned} U_C(s,A) = \;&\mathbf{if}\; s = \langle\,\rangle \\ &\mathbf{then}\; (a : A \to w \to SKIP) \\ &\mathbf{else}\; (w \to SKIP \;\Box\; head(s) \to U_C(tail(s), A)) \end{aligned}$$

3. *For* $s \in \alpha(SPEC)^*$, *define* Divergence Tests $U_D(s)$ *by*

$$\begin{aligned} U_D(s) = \;&\mathbf{if}\; s = \langle\,\rangle \\ &\mathbf{then}\; w \to SKIP \\ &\mathbf{else}\; (w \to SKIP \;\Box\; head(s) \to U_D(tail(s))) \end{aligned}$$

4. *For a sequence of events* $s \in \alpha(SPEC)^*$, *define* Robustness Tests $U_R(s)$ *by*

$$\begin{aligned} U_R(s) = \;&\mathbf{if}\; s = \langle\,\rangle \\ &\mathbf{then}\; w \to SKIP \\ &\mathbf{else}\; head(s) \to U_R(tail(s)) \end{aligned}$$

□

Definition 4 is motivated by the following lemma:

Lemma 13 *In the sense of Definition 2, the Hennessy Test Cases detect the following failures:*

1. $U_S(s,a)$ *detects safety failure* $s^\frown\langle a\rangle$.
2. $U_C(s,A)$ *detects requirements coverage failure* (s,A).
3. $U_D(s)$ *detects divergence failure* s.
4. $U_R(s)$ *detects robustness failure* s.

□

Note that the Hennessy test classes even *characterise* the associated failure types: If $s^\frown\langle a\rangle \notin \mathit{Traces}(P)$ then $P\ \underline{\mathit{must}}\ U_S(s,a)$ follows. Analogous results hold for $U_C(s,A)$, $U_D(s)$, $U_R(s)$. In our context $s \in \mathit{Div}(P)$ means $P/s = \mathit{CHAOS}$ in the sense of [44], that is, P/s may both *diverge internally (livelock)* and produce and refuse arbitrary *external* events. The tests $U_D(s)$ have been designed by Hennessy to detect internal divergence only. Conversely, the tests $U_S(s,a)$ and $U_C(s,A)$ detect external chaotic behaviour but cannot distinguish internal divergence from deadlock. However, using the three test classes together enables us to distinguish deadlock, livelock and external chaotic behaviour. Note that $P\ \underline{\mathit{must}}\ U_S(s,a)$ also implies $s \notin \mathit{Div}(P)$, because divergence along s would imply that every continuation of s, specifically $s^\frown\langle a\rangle$ would be a trace of P. $P\ \underline{\mathit{must}}\ U_C(s,A)$ implies $s \notin \mathit{Div}(P)$, because divergence along s implies the possibility to refuse every subset of $\alpha(P)$ after s. These observations are summarised in the following lemma:

Lemma 14 *Let P a CSP process, $A \subseteq \alpha(P)$ with $A \neq \emptyset$, $a \in \alpha(P)$, $s \in \alpha(P)^*$. Then*

1. $P\ \underline{\mathit{must}}\ U_S(s,a)$ *iff* $s^\frown\langle a\rangle \notin \mathit{Traces}(P)$.
2. $P\ \underline{\mathit{must}}\ U_C(s,A)$ *iff* $(s,A) \notin \mathit{Fail}(P)$.
3. $P\ \underline{\mathit{must}}\ U_D(s)$ *iff* $s \notin \mathit{Div}(P)$.
4. $P\ \underline{\mathit{may}}\ U_R(s)$ *iff* $s \in \mathit{Traces}(P)$.
5. $P\ \underline{\mathit{must}}\ U_S(s,a)$ *implies* $P\ \underline{\mathit{must}}\ U_D(s)$.
6. $P\ \underline{\mathit{must}}\ U_C(s,A)$ *implies* $P\ \underline{\mathit{must}}\ U_D(s)$.

□

Now Hennessy's results about the relation between testing and refinement can be re-phrased for our context as follows:

Theorem 7 *The classes $U_S(s,a)$, $U_C(s,A)$, $U_D(s)$, $U_R(s)$ of test cases are related to T, F, FD, FDR-refinement as follows:*

1. *If* $\mathit{SPEC}\ \underline{\mathit{must}}\ U_S(s,a)$ *implies* $\mathit{IMP}\ \underline{\mathit{must}}\ U_S(s,a)$ *for all* $a \in \alpha(\mathit{SPEC})$, $s \in \alpha(\mathit{SPEC})^*$, *then* $\mathit{SPEC} \sqsubseteq_T \mathit{IMP}$.
2. *If* $\mathit{SPEC} \sqsubseteq_T \mathit{IMP}$ *and* $\mathit{SPEC}\ \underline{\mathit{must}}\ U_C(s,A)$ *implies* $\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A)$ *for all* $s \in \alpha(\mathit{SPEC})^*$, $A \subseteq \alpha(\mathit{SPEC})$, *then* $\mathit{SPEC} \sqsubseteq_F \mathit{IMP}$.

3. *If* $SPEC \sqsubseteq_F IMP$ *and* $SPEC$ $\underline{must}$ $U_D(s)$ *implies* IMP $\underline{must}$ $U_D(s)$ *for all* $s \in \alpha(SPEC)^*$, *then* $SPEC \sqsubseteq_{FD} IMP$.
4. *If* $SPEC \sqsubseteq_{FD} IMP$ *and* $SPEC$ $\underline{may}$ $U_R(s)$ *implies* IMP $\underline{may}$ $U_R(s)$ *for all* $s \in \alpha(SPEC)^*$, *then* $SPEC \sqsubseteq_{FDR} IMP$.

Proof.
The theorem follows directly from Lemma 14 and the definition of T, F, FD, FDR-refinement.
□

The implications in this theorem become equivalences if additionally $Div(IMP) \subseteq Div(SPEC)$ holds. Theorem 7 shows that "in principle" only requirements-driven test design is needed: It is only necessary to execute test cases that will succeed for $SPEC$. On the other hand, the properties covered by Theorem 7 cannot be verified by means of black-box tests alone, because they require the analysis of *every* possible execution. If IMP is nondeterministic, it may possibly refuse in every test execution to engage into the trace leading to the detection of an error which could be uncovered by the test cases $U_S(s,a)$, $U_C(s,A)$, $U_D(s)$. Conversely, for the test of robustness, the refusal to engage into trace s in a test execution does not prove that $s \notin Traces(IMP)$. Therefore in general, a *test monitor* collecting internal IMP-information about the executions performed will be mandatory to determine whether "sufficient" test executions have been performed. Note that this is no disadvantage of the defined classes of tests but inherent in every testing approach that is sensitive to nondeterminism.

Trustworthy Collections of Test Cases

When performing a test suite to investigate the correctness properties of a system, a crucial objective is to exercise "as few test cases as possible". Therefore the main result (Theorem 8) of this section is devoted to the characterisation of subsets of Hennessy test cases which still suffice to establish the refinement properties implied by Theorem 7. Theorem 9 shows that the sets defined for $\sqsubseteq_T$- and $\sqsubseteq_F$-test in Theorem 8 are even minimal. For non-terminating processes, minimal sets of test cases investigating divergence and robustness properties cannot be defined. However, Theorem 8 shows that they can at least be chosen considerably smaller than the collection used in Theorem 7.

Before presenting the theorems, two additional definitions for the classification of test cases are introduced:

- *Meaningful* collections of test cases against a given specification $SPEC$ are those which succeed when executed in parallel with $SPEC$ itself. Since in most applications exhaustive testing will be impossible, it would be misleading if test cases U not fulfilled by $SPEC$ were executed, because they would add nothing to the question of implementation correctness while giving users the impression that – in case of successful execution – "some progress" had been made.
- *Trustworthy* collections of test cases have the additional property to establish a refinement relationship between $SPEC$ and its implementation IMP, when exercised completely and successfully on IMP. Conversely speaking, trustworthy collections contain for each implementation error a test case which is able to uncover this error.

Definition 5 *A collection $\mathcal{U}$ of admissible test cases is called*

1. meaningful for the $\underline{may}$-test against $SPEC$, *iff*

$$(\forall\, U : \mathcal{U} \bullet s^\frown\langle w\rangle \in Traces(U) \Rightarrow s \in Traces(SPEC))$$

2. meaningful for the $\underline{must}$-test against $SPEC$, *iff it is meaningful for the $\underline{may}$-test and*

$$(\forall\, U : \mathcal{U} \bullet SPEC\ \underline{must}\ U)$$

□

Observe that *"meaningful for the $\underline{may}$-test against SPEC"* requires more than just $(\forall\, u : \mathcal{U} \bullet SPEC\ \underline{may}\ U)$: The tests of a meaningful collection may only show success after executions that can also be performed by $SPEC$.

Example 4.5 Let

$$SPEC = a \to STOP$$
$$IMP = b \to STOP$$
$$U = a \to w \to SKIP \;\square\; b \to w \to SKIP$$

Then $SPEC\ \underline{must}\ U$, but U is not meaningful for the test against $SPEC$, since it also signals success when exercised on implementations IMP violating the safety requirement $SPEC \sqsubseteq_T IMP$.
□

The notion of *Trustworthiness* is now defined by

Definition 6 *Let $\mathcal{U}$ be a set of admissible test cases for the test against SPEC with $\mathcal{U} = \mathcal{U}_T \cup \mathcal{U}_F \cup \mathcal{U}_{FD} \cup \mathcal{U}_R$, such that $\mathcal{U}_T, \mathcal{U}_F, \mathcal{U}_{FD}$ contain meaningful $\underline{must}$-tests and $\mathcal{U}_R$ contains meaningful $\underline{may}$-tests.*

1. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_T$-tests *iff*

$$(\forall\, U : \mathcal{U}_T \bullet IMP\ \underline{must}\ U) \Rightarrow SPEC \sqsubseteq_T IMP$$

2. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_F$-tests *iff it is trustworthy for $\sqsubseteq_T$-tests and*

$$(\forall\, U : \mathcal{U}_F \bullet IMP\ \underline{must}\ U) \Rightarrow SPEC \sqsubseteq_F IMP$$

3. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_{FD}$-tests *iff it is trustworthy for $\sqsubseteq_F$-tests and*

$$(\forall\, U : \mathcal{U}_{FD} \bullet IMP\ \underline{must}\ U) \Rightarrow SPEC \sqsubseteq_{FD} IMP$$

4. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_{FDR}$-tests *iff it is trustworthy for $\sqsubseteq_{FD}$-tests and*

$$(\forall\, U : \mathcal{U}_R \bullet IMP\ \underline{may}\ U) \Rightarrow SPEC \sqsubseteq_{FDR} IMP$$

□

Definition 5 defines a considerable reduction of the test classes, when compared to the preconditions of Theorem 7: In the theorem, the Hennessy Test Cases were selected for *every* sequence $s \in \alpha(SPEC)^*$ and *every* set $A \subseteq \alpha(SPEC)$ of events. However, the following theorem characterises the subset of Hennessy Test Cases introduced above which is "relevant" for the verification of refinement properties. It shows that the substantial restrictions of the test cases to be executed according to Definition 5 are really sensible. A more detailed interpretation of the theorem is given at the end of this section.

Theorem 8 *For a given specification SPEC, define collections of test cases by*

1. $\mathcal{H}_S(SPEC) = \{U_S(s,a) \mid s \in Traces(SPEC) - Div(SPEC) \wedge a \notin [SPEC/s]^0\}$
2. $$\begin{aligned}\mathcal{H}_C(SPEC) = \{U_C(s,A) \mid\ & s \in Traces(SPEC) - Div(SPEC) \wedge \\ & A \subseteq [SPEC/s]^0 \wedge \\ & (\forall\, R : Ref(SPEC/s) \bullet A \not\subseteq R) \wedge \\ & (\forall\, X : \mathbb{P}\, A - \{A\} \bullet (\exists\, R : Ref(SPEC/s) \bullet X \subseteq R))\}\end{aligned}$$
3. $$\begin{aligned}\mathcal{H}_D(SPEC) = \{U_D(s) \mid\ & s \in Traces(SPEC) - Div(SPEC) \wedge \\ & (\forall\, u : Traces(SPEC) - Div(SPEC) \bullet \\ & \quad s \leq u \wedge [SPEC/u]^0 = \emptyset \Rightarrow s = u)\}\end{aligned}$$
4. $$\begin{aligned}\mathcal{H}_R(SPEC) = \{U_R(s) \mid\ & s \in Traces(SPEC) \wedge \\ & (\forall\, u : Traces(SPEC) \bullet \\ & \quad s \leq u \wedge [SPEC/u]^0 = \emptyset \Rightarrow s = u)\}\end{aligned}$$

Then

1. *$\mathcal{H}_S(SPEC)$ is trustworthy for $\sqsubseteq_T$-tests.*
2. *$\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC)$ is trustworthy for $\sqsubseteq_F$-tests.*
3. *$\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC) \cup \mathcal{H}_D(SPEC)$ is trustworthy for $\sqsubseteq_{FD}$-tests.*
4. *$\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC) \cup \mathcal{H}_D(SPEC) \cup \mathcal{H}_R(SPEC)$ is trustworthy for $\sqsubseteq_{FDR}$-tests.*

Proof.
We prove 2, the other cases are shown analogously. The proof is structured as follows: Our first objective is to show that the test cases $U_C(s,A) \in \mathcal{H}_C(SPEC)$ are meaningful for the $\underline{must}$-test against *SPEC* in the sense of Definition 5. By definition of $\mathcal{H}_C(SPEC)$, these test cases only execute traces of *SPEC* before successful termination, so they are obviously meaningful for the $\underline{may}$-test against *SPEC*. It remains to show that *SPEC* $\underline{must}$ $U_C(s,A)$ for these test cases. This is demonstrated in Proof Obligation 2.1. Next, we wish to show that the test cases in $\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC)$ are powerful enough to detect every possible violation of $\sqsubseteq_F$-refinement. From Part 1 of the theorem we know that the tests of $\mathcal{H}_S(SPEC)$ suffice to detect every violation of $\sqsubseteq_T$-refinement. It remains to show that the tests of $\mathcal{H}_C(SPEC)$ detect missing requirements coverage. Now Theorem 7 implies that the test $U_C(s,A)$ are at least capable for this task if we consider *all* traces $s \in \alpha(SPEC)^*$ and sets of events $A \subseteq \alpha(SPEC)$. We will therefore prove that whenever an *arbitrary* $\underline{must}$-test $U_C(s,A)$ satisfied by *SPEC* fails for an implementation *IMP*, there will also be a test *contained in* $\mathcal{H}_C(SPEC)$ which fails for *IMP*, too. Now the collection of all test cases $U_C(s,A)$ for which *SPEC* $\underline{must}$ $U_C(s,A)$ holds can be divided into two groups, depending

on whether s is a trace of $SPEC$ or not. The former case is handled by Proof Obligation 2.2, the latter by Proof Obligation 2.3.

Proof Obligation 2.1. For $U_C(s, A) \in \mathcal{H}_C$, show that $SPEC\ \underline{must}\ U_C(s, A)$.

According to the definition of $\mathcal{H}_C$, we have $s \in Traces(SPEC) - Div(SPEC)$. Moreover, the definitions of $\mathcal{H}_C$ and $U_C(s, A)$ imply that every execution of $(U_C(s, A) \parallel SPEC)$ is a prefix of some $u^\frown\langle w\rangle$, where u is a prefix of $s^\frown\langle a\rangle$ with $a \in A$.

Case 2.1.1. Suppose, an execution of $(U_C(s, A) \parallel SPEC)$ does not reach $last(s)$. Since neither $U_C(s, A)$ nor $SPEC$ diverge along s, the same holds for $(U_C(s, A) \parallel SPEC)$, therefore $SPEC$ must block before $last(s)$ is reached, and as a consequence $(U_C(s, A) \parallel SPEC)$ is ready to produce any event not contained in $\alpha(SPEC)$. Since the only event not contained in $\alpha(SPEC)$ and executable in this state is w, this leads to successful execution of $(U_C(s, A) \parallel SPEC)$.

Case 2.1.2. Suppose $last(s)$ is reached in an execution of $(U_C(s, A) \parallel SPEC)$. Since $SPEC$ does not diverge along s, application of the Normal Form Theorem 12 yields

$$SPEC/s = \sqcap_{\,R:Ref(SPEC/s)}(x : ([SPEC/s]^0 \setminus R) \to SPEC/s^\frown\langle x\rangle)$$

Since $A \subseteq [SPEC/s]^0$ and $A \not\subseteq R$ holds for every refusal R of $SPEC/s$, we get

$$(\forall\, R : Ref(SPEC/s) \bullet (\exists\, a_R : A \bullet a_R \in ([SPEC/s]^0 \setminus R)))$$

Together with the normal form representation, this shows that $SPEC/s$ cannot refuse all events of A. As a consequence the execution also yields success w. Case 2.1.1 and 2.1.2 prove that $SPEC\ \underline{must}\ U_C(s, A)$. This shows the correctness of Proof Obligation 2.1.

Proof Obligation 2.2. Show that for all $s_1 \in Traces(SPEC)$

$$\begin{array}{l}(\forall\, A_1 : \mathbb{P}\,\alpha(SPEC) \bullet \\ \qquad SPEC\ \underline{must}\ U_C(s_1, A_1) \wedge \neg\,(IMP\ \underline{must}\ U_C(s_1, A_1)) \Rightarrow \\ \qquad\qquad (\exists\, A : \mathbb{P}\,\alpha(SPEC) \bullet U_C(s_1, A) \in \mathcal{H}_C \wedge \neg\,(IMP\ \underline{must}\ U_C(s_1, A_1))))\end{array}$$

holds. This means that whenever IMP contains a requirements coverage failure (s_1, A_1) with $s_1 \in Traces(SPEC)$ and arbitrary A_1, it will also contain a requirements coverage failure (s_1, A) which can be detected by a test $U_C(s_1, A) \in \mathcal{H}_C$.

Assume

$$s_1 \in Traces(SPEC) \wedge SPEC\ \underline{must}\ U_C(s_1, A_1) \wedge U_C(s_1, A_1) \notin \mathcal{H}_C \wedge \neg\,(IMP\ \underline{must}\ U_C(s_1, A_1))$$

Then, because of the structure of $U_C(s_1, A_1)$, two situations may be the cause for the failure of IMP:

1. $s_1 \in Div(IMP)$
2. $s_1 \in Traces(IMP) - Div(IMP) \wedge (\exists\, R : Ref(IMP/s_1) \bullet A_1 \subseteq R)$

In both cases we have $s_1 \in Traces(IMP)$. Because

$$s_1 \in Traces(SPEC) \wedge SPEC\ \underline{must}\ U_C(s_1, A_1)$$

$A_1 \cap [SPEC/s]^0$ cannot be completely contained in any of the $R \in Ref(SPEC/s)$. Let $A \subseteq A_1 \cap [SPEC/s]^0$ be one of the smallest sets such that $(\forall R : Ref(SPEC/s) \bullet A \not\subseteq R)$. Then $U_C(s_1, A) \in \mathcal{H}_C$ and $SPEC\ \underline{must}\ U_C(s_1, A)$.

Case 2.2.1. If $s_1 \in Div(IMP)$ is the cause for $\neg\ (IMP\ \underline{must}\ U_C(s_1, A_1))$, then also $\neg\ (IMP\ \underline{must}\ U_C(s_1, A))$, because divergence along s_1 will lead to an execution of $(U_C(s_1, A) \parallel IMP)$ where any visible event is refused before success w could be signalled.

Case 2.2.2. If $s_1 \in Traces(IMP) - Div(IMP) \wedge (\exists R : Ref(IMP/s_1) \bullet A_1 \subseteq R)$ is the cause for $\neg\ (IMP\ \underline{must}\ U_C(s_1, A_1))$, then also $A \subseteq A_1$ will be contained in such an R. This yields once more $\neg\ (IMP\ \underline{must}\ U_C(s_1, A))$, because an execution of $(U_C(s_1, A) \parallel IMP)$ will block A-events after s_1, and $U_C(s_1, A)/s_1$ can signal success w only after having first engaged into some $a \in A$.

Proof Obligation 2.3. Show that for all $s_1 \notin Traces(SPEC)$

$$
\begin{array}{l}
(\forall A_1 : \mathbb{P}\,\alpha(SPEC) \bullet \\
\quad \neg\ (IMP\ \underline{must}\ U_C(s_1, A_1)) \Rightarrow \\
\qquad (\exists s : Traces(SPEC);\ a : \alpha(SPEC) \bullet U_S(s, a) \in \mathcal{H}_S \wedge \neg\ (IMP\ \underline{must}\ U_S(s, a))))
\end{array}
$$

holds. To understand the objective of this proof obligation, recall that $SPEC\ \underline{must}\ U_C(s_1, A_1)$ holds for $s_1 \notin Traces(SPEC)$ and any A_1: $SPEC$ cannot diverge after a prefix u of s_1 because otherwise any extension of u would be contained in $Traces(SPEC)$. As a consequence, $SPEC$ will block after a prefix u of s_1 and $U_C(s_1, A_1)/u$ is forced to take the branch $w \rightarrow SKIP$. Conversely, $\neg\ (IMP\ \underline{must}\ U_C(s_1, A_1))$ implies that $s_1 \in Traces(IMP)$.

Since $s_1 \notin Traces(SPEC)$ there exists s_{11}, s_{12} such that $s_1 = s_{11}{}^\frown s_{12} \wedge s_{11} \in Traces(SPEC) \wedge head(s_{12}) \notin [SPEC/s_{11}]^0$. As a consequence, $SPEC\ \underline{must}\ U_S(s_{11}, head(s_{12}))$, but $s_1 \in Traces(IMP)$ implies $\neg\ (IMP\ \underline{must}\ U_S(s_{11}, head(s_{12}))$. Since $U_S(s_{11}, head(s_{12})) \in \mathcal{H}_S$, this proves Obligation 2.3 and completes the proof of 2.
□

The following Theorem shows that the collections $\mathcal{H}_S$ and $\mathcal{H}_C$ are minimal:

Theorem 9 *The collections $\mathcal{H}_S, \mathcal{H}_C$ of test cases defined in Theorem 8 are minimal in the following sense:*

1. *If $\mathcal{H} \subset \mathcal{H}_S$ there exists a process P satisfying $P\ \underline{must}\ U$ for all $U \in \mathcal{H}$ but not refining $SPEC$ in the trace model.*
2. *If $U_C(s, A) \in \mathcal{H}_C$ and $B \subset A$ then $\neg\ (SPEC\ \underline{must}\ U_C(s, B))$.*
3. *If $\mathcal{H} \subset \mathcal{H}_C$ there exists a process P satisfying $P\ \underline{must}\ U$ for all $U \in \mathcal{H}_S \cup \mathcal{H}$ but not refining $SPEC$ in the failures model.*

Proof.

Proof of 1. Suppose that $U_S(s,a) \in \mathcal{H}_S$ and $\mathcal{H} = \mathcal{H}_S - \{U_S(s,a)\}$. Define

$$\alpha(P) = \alpha(SPEC)$$

$$Traces(P) = Traces(SPEC) \cup \{s^\frown\langle a\rangle\}$$

$$\begin{aligned} P/u = \textbf{if } & u = s \\ & \textbf{then } (x : [SPEC/u]^0 \to P/u^\frown\langle x\rangle \mathbin{\Box} a \to STOP) \\ & \textbf{else } (x : [SPEC/u]^0 \to P/u^\frown\langle x\rangle) \end{aligned}$$

The $P \,\underline{must}\, U$ holds for all $U \in \mathcal{H}$, but P violates the safety requirement $s^\frown\langle a\rangle \notin Traces(P)$.

Proof of 2. Suppose $U_C(s,A) \in \mathcal{H}_C$ and $B \subset A$. The definition of $\mathcal{H}_C$ implies the existence of a refusal $R \in Ref(SPEC/s)$ such that $B \subseteq R$. As a consequence there exists an execution $(U_C(s,B) \parallel SPEC)$ blocking and consequently failing after trace s. Therefore, $\neg\,(SPEC \,\underline{must}\, U_C(s,B))$ holds, which proves (2).

Proof of 3. Let $\{U_C(s,A_1),\ldots,U_C(s,A_n)\} \subseteq \mathcal{H}_C$ be an enumeration of the $\underline{must}$-tests in $\mathcal{H}_C$ to be executed for fixed trace s. Suppose $\mathcal{H} = \mathcal{H}_C - \{U_C(s,A_n)\}$. Then define a CSP process P by

$$\alpha(P) = \alpha(SPEC)$$

$$Traces(P) = Traces(SPEC)$$

$$\begin{aligned} P/u = \textbf{if } & u = s \\ & \textbf{then } (x : ([SPEC/u]^0 \setminus A_n) \to P/u^\frown\langle x\rangle \mathbin{\Box} ((x : A_n \to P/u^\frown\langle x\rangle) \sqcap STOP)) \\ & \textbf{else } (x : [SPEC/u]^0 \to P/u^\frown\langle x\rangle) \end{aligned}$$

Since $Traces(P) = Traces(SPEC)$ by construction, $P \,\underline{must}\, U$ holds for all safety tests in $\mathcal{H}_S$. For traces $u \neq s$, $P \,\underline{must}\, U_C(u,A)$ holds for all tests in $\mathcal{H}_C$, because P/u never refuses an event that might be accepted by $SPEC/u$. For $u = s$, observe that the sets A_i are minimal in the sense of part (2) of the theorem, and therefore $A_i \setminus A_n \neq \emptyset$ holds for all $i = 1,\ldots,(n-1)$. As a consequence, $[SPEC/u]^0 \setminus A_n$ contains at least one A_i-event for all $i = 1,\ldots,(n-1)$. This implies $P \,\underline{must}\, U_C(s,A_i)$ for $i = 1,\ldots,(n-1)$, so $P \,\underline{must}\, U$ for all tests $U \in \mathcal{H}$. However, P/s may refuse A_n completely, while $SPEC/s$ will always accept at least one A_n-event. As a consequence, P does not refine $SPEC$ in the failures model.
□

Interpretation of Theorem 8 and Theorem 9 Let us first state the obvious: For terminating systems, refinement properties can be verified by means of a finite number of tests, because the trace space is finite and we always assume finite alphabets, hence the sets $\mathcal{H}_S, \mathcal{H}_C, \mathcal{H}_D, \mathcal{H}_R$ will also be finite.

The definitions of $\mathcal{H}_S, \mathcal{H}_C, \mathcal{H}_D$ indicate further that it is not necessary to perform any tests for traces s after which $SPEC$ diverges[8], since in such a case $SPEC/s$ will allow chaotic

[8]Of course, it is questionable if specifications allowing divergence will be used in practice at all.

behaviour which does not restrict the admissible behaviour of IMP/s. For the test of safety properties, the definition of $\mathcal{H}_S$ states that we only have to use those test cases $U_S(s, a)$, where s is a trace of *SPEC*, but *SPEC*$/s$ does not admit event a. For the requirements coverage tests $U_C(s, A)$, $\mathcal{H}_C$ indicates that only the smallest sets A, such that *SPEC*$/s$ can never refuse A completely, have to be tested. As a consequence, it is unnecessary to exercise any tests $U_C(s, A)$, if *SPEC*$/s$ may refuse the full alphabet, e. g., if $SPEC/s = P \sqcap STOP$. For implementation purposes it is useful to observe that since $Ref(SPEC/s)$ is always finite and subset-closed, we only need to evaluate the *maximal* refusals $R \in refMax(SPEC/s)$ to compute the sets A satisfying $(\forall R : Ref(SPEC/s) \bullet A \not\subseteq R)$.

The definitions of $\mathcal{H}_D$ and $\mathcal{H}_R$ are motivated by the fact that for the test of divergence and robustness properties we only have to analyse *maximal* traces: If *SPEC* terminates or blocks after a trace u, the tests corresponding to proper prefixes of u are covered by $U_D(u)$ and $U_R(u)$, so only the latter are contained in $\mathcal{H}_D$ and $\mathcal{H}_R$ respectively.

Observe that the collections of tests introduced in Theorem 8 completely characterise the refinement relations between *SPEC* and *IMP*: Any *IMP* refining *SPEC* must satisfy all corresponding Hennessy tests U_S, U_C, U_D, U_R, and the collections $\mathcal{H}_S, \mathcal{H}_C, \mathcal{H}_D, \mathcal{H}_R$ are subsets of these Hennessy tests.

Theorem 9 shows that $\mathcal{H}_S$ and $\mathcal{H}_C$ are indeed minimal sets for T- and F-tests against *SPEC*: If one test $U_S(s, a)$ is removed from $\mathcal{H}_S$, a process with safety failure $s \frown \langle a \rangle$ could be constructed, for which all the remaining tests would succeed. For each $U_C(s, A)$ in $\mathcal{H}_C$, the set A cannot be reduced, otherwise the test would no longer succeed for *SPEC*. Moreover, removing a test $U_C(s, A)$ from $\mathcal{H}_C$ would admit processes P satisfying the remaining tests without refining *SPEC* in the failures model. $\mathcal{H}_D$ and $\mathcal{H}_R$, however, cannot be defined as minimal sets, as soon as *SPEC* describes a non-terminating system: If $comp = \langle s(1), s(2), s(3), \ldots \rangle$ is an infinite computation of *SPEC*, $\mathcal{H}_D$ and $\mathcal{H}_R$ must contain infinitely many tests associated with prefixes $s_1 < s_2 < s_3 < \ldots$ of *comp*, and each infinite subset of these tests would suffice to verify correct behaviour along *comp*. At least we can state that any $\mathcal{H}_D^0 \subseteq \mathcal{H}_D$ satisfying

$$(\forall u : Traces(SPEC) - Div(SPEC) \bullet \exists s : \mathcal{H}_D^0 \bullet u \leq s)$$

is sufficient to detect divergence failures against *SPEC* and any $\mathcal{H}_R^0 \subseteq \mathcal{H}_R$ satisfying

$$(\forall u : Traces(SPEC) - Div(SPEC) \bullet \exists s : \mathcal{H}_R^0 \bullet u \leq s)$$

is sufficient to detect robustness failures.

4.5.3 Trustworthy Test Drivers

The Concept of Test Drivers

Test Drivers are hardware and software devices controlling the executions of test cases for a target system. To formalise this notion, recall that a *context* is a CSP term $\mathcal{C}(X)$ with one free identifier X, so every context can be regarded as a continuous function mapping CSP processes X to CSP processes $\mathcal{C}(X)$ (see [71, p.160]). Apart from the free identifier X, $\mathcal{C}(X)$ may contain other CSP processes as parameters.

Definition 7 *A* Test Driver for the test against *SPEC is a context* $\mathcal{D}(X)$ *using admissible test cases* U_i *satisfying* $\alpha(SPEC) = \alpha(U_i) \setminus \{w\}$ *as parameters.*
□

Example 4.6 Every admissible test case U can be regarded as a test driver $X \mapsto (U \parallel X)$ executing only a single test case.
□

Example 4.7 We will focus on test drivers of the form

$$\mathcal{D}(X) = (i := 0);\ *(U_i \parallel X\hat{}(w \rightarrow monitor?next \rightarrow (\textbf{if}\ next\ \textbf{then}\ i := i+1;\ SKIP\ \textbf{else}\ SKIP)));$$

with admissible test cases U_i. A test driver of this type will execute the test cases in a certain ordering $U_1, U_2, \ldots$; one test case at a time and with only one copy of the target system $X = IMP$ running. As soon as a test case signals success w, the execution will be interrupted. An input *monitor?next* will be required from a process monitoring the test coverage achieved so far with the actual test U_i. If the monitor signals $next = true$, the next test case U_{i+1} will be performed, otherwise U_i will be repreated. If U_i is a $\underline{may}$-test, *next* is always set to *true*. The problem how to determine test coverage can be separated from the task of test generation, execution and evaluation. In principle, the monitor could also be a person, manually interacting with the test driver.
□

The minimum requirement for a *trustworthy* test driver is given by the following definition:

Definition 8 *A test driver* $\mathcal{D}(X)$ *is* meaningful for the test against *SPEC*, *iff the test cases* U_i *appearing as parameters of* $\mathcal{D}(X)$ *are taken from a collection* $\mathcal{U}$ *of meaningful* $\overline{may}$ *- and* $\underline{must}$ *-tests against SPEC in the sense of Definition 5.*
□

Definition 8 requires that the test driver should only perform executions of test cases which may succeed for *SPEC*. This will ensure that each successful test execution performed by $\mathcal{D}(IMP)$ will at least indicate that the implementation "can perform something which is correct according to the specification". However, this minimum requirement does not ensure that any kind of "progress" is made during the test: $\mathcal{D}(IMP)$ might perform always the same test execution, so that an additional success indication would not contribute any new insight about the implementation. Therefore we introduce

Definition 9 *Let* $\mathcal{D}(X)$ *be a meaningful test driver for the test against SPEC, performing test cases of a collection* $\mathcal{U}$ *in the order* $U_1, U_2, U_3, \ldots$. *Let* $\sqsubseteq \in \{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}, \sqsubseteq_{FDR}\}$. *Then* $\mathcal{D}(X)$ *is called* trustworthy for $\sqsubseteq$-test against *SPEC*, *iff the following conditions hold:*

- $\mathcal{U}$ *contains a subset* $\mathcal{U}_{\sqsubseteq}$ *which is trustworthy for* $\sqsubseteq$ *-test against SPEC according to Definition 6.*
- *For every safety-, requirements coverage-, divergence- or robustness-failure violating* $\sqsubseteq$, *there exists an* $n \in \mathbb{N}$ *such that* $U_n \in \mathcal{U}_{\sqsubseteq}$ *detects this failure in the sense of Definition 2.*

□

Definition 9 covers the intuitive understanding of trustworthiness in a formal way: whenever *IMP* may perform a failure, this can be uncovered by a test case which is guaranteed to be chosen by the driver after having selected a finite number of other test cases.

Example 4.8 A test driver performing Hennessy Test Cases $U_x(s, \ldots)$ according to a depth-first ordering of the traces s is *not* trustworthy in the sense of Definition 9: For recursive implementations the test case selection may follow an infinite sequence of events, so that an infinite number of test cases may precede a test case which could detect a failure by following a "neighbouring" trace.
□

Theorem 10 *Every test driver*

$$\begin{aligned}\mathcal{D}(X) = (i := 0);\ *(U_i \parallel X\hat{}(w \rightarrow & \mathit{monitor}?\mathit{next} \\ & \rightarrow (\textbf{if } \mathit{next} \textbf{ then } i := i+1;\ \mathit{SKIP} \textbf{ else } \mathit{SKIP})));\end{aligned}$$

applying the Hennessy Tests

$$U_i \in \mathcal{H}_S(\mathit{SPEC}) \cup \mathcal{H}_C(\mathit{SPEC}) \cup \mathcal{H}_D(\mathit{SPEC}) \cup \mathcal{H}_R(\mathit{SPEC})$$

according to Theorem 8, ordered by the length of the defining traces, is trustworthy for $\sqsubseteq_{FDR}$ *-test against SPEC.*

Proof.
Since $\alpha(\mathit{SPEC})$ is finite, every $[\mathit{IMP}/s]^0$ is also finite. As a consequence, $\mathit{Traces}(\mathit{IMP})$ contains only a finite number of traces with fixed length $n \in \mathbb{N}$. Since also $\mathit{refMax}(\mathit{IMP}/s)$ is always finite, the number of Hennessy Tests $U_i \in \mathcal{H}$ with defining trace of length less or equal n will also be finite.

Since according to Theorem 8 the test cases of $\mathcal{H}$ detect every type of failure and every failure occurs after a finite trace, the theorem follows.
□

Analogous results hold for $\sqsubseteq_T$, $\sqsubseteq_F$, $\sqsubseteq_{FD}$ -test.

Test Drivers for Reactive Systems

The testing methodology presented so far in this chapter will now be specialised on the development of test drivers for the automated test of *Reactive Systems.*

Separation of Environment and Target System In the context of reactive systems it is useful to distinguish between the target system and its operational environment in an explicit way, when investigating properties of a specification *SPEC* and implementation *IMP*: The very purpose of reactive systems is to interact continuously with their environment. In many applications certain hypotheses are made about the environment behaviour. This means that the target system is not expected to act properly in *every* context. Indeed, the objective of the test suite is to *ensure the correct behaviour of the target system when running in an operational environment satisfying these hypotheses.* Therefore test drivers should

- *simulate* the environment behaviour
- *test* the target system behaviour in the simulated environment

To formalise the notion of an operational environment we will consider expressions of the type

$$SPEC = \mathcal{E}(ASYS) \setminus (\alpha(\mathcal{E}(ASYS)) - I)$$

with the following interpretation: $\mathcal{E}(X)$ is a context and $ASYS$ is the abstract specification of the target system to be developed. The processes appearing as parameters in $\mathcal{E}$ represent the operational environment. The correctness of a reactive system implementation will only be decided with respect to a subset I of observables considered as relevant from the point of view of the application. Therefore the specification consists of $\mathcal{E}(ASYS)$ with all events apart from I concealed. The implementation will be denoted by

$$IMP = \mathcal{E}(SYS) \setminus (\alpha(\mathcal{E}(SYS)) - I)$$

where SYS is the target system "plugged into" environment $\mathcal{E}$. It is natural to demand that $I \subseteq \alpha(\mathcal{E}(ASYS)) \cap \alpha(\mathcal{E}(SYS))$.

Note that the alphabets of the abstract specification $ASYS$ and the implementation SYS of the target system may be distinct, though – as required according to the preceding sections – $\alpha(SPEC) = \alpha(IMP)$. Of course, the verification obligations for IMP with respect to $SPEC$ remain as introduced above, using *T,F,FD,FDR*-refinement.

In many applications, the configuration of a reactive system and its environment will be appropriately described by the following definition:

Definition 10 *A* standard configuration $(E, ASYS, SYS, I)$ (for reactive systems) *consists of CSP processes $E, ASYS, SYS$ and a set of events I such that*

$$I = \alpha(E) \cap \alpha(ASYS) = \alpha(E) \cap \alpha(SYS)$$

For a given standard configuration, the context $\mathcal{E}_0(X) = (E \parallel X)$ is called the environment.

$$SPEC = \mathcal{E}_0(ASYS) \setminus (\alpha(\mathcal{E}_0(ASYS)) - I)$$

is called the specification*,*

$$IMP = \mathcal{E}_0(SYS) \setminus (\alpha(\mathcal{E}_0(SYS)) - I)$$

the implementation. *For $\sqsubseteq \in \{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}, \sqsubseteq_{FDR}\}$, a standard configuration is called* $\sqsubseteq$-correct*, if $SPEC \sqsubseteq IMP$ holds.*
□

Note that it is not a severe restriction to require that the set I of observables should be identical to the set of events shared between environment and target system: Since in the context of reactive systems the tester or any other observer is a part of the environment, any observable event generated by the target system must be shared with the environment. Moreover, if we wish to observe environment events e during test execution that do not occur at the interface to the target system, but "inside" the environment, the standard configuration $(E, ASYS, SYS, I)$ can be re-arranged, such that the new target system is extended by auxiliary system components RUN_e, running in interleaving mode with the "real" target system and sharing e with the environment without ever refusing its occurrence: The new configuration would be defined by

$$(E', ASYS', SYS', I') = (E, (ASYS \mathbin{|||} RUN_e), (SYS \mathbin{|||} RUN_e), I \cup \{e\})$$

We will use the abbreviation $P_I = P \setminus (\alpha(P) \setminus I)$. Note that in a standard configuration $(E \parallel ASYS)_I = (E_I \parallel ASYS_I)$ and $(E \parallel SYS)_I = (E_I \parallel SYS_I)$ holds, because the hiding operator distributes through $\parallel$, if none of the interface events shared between the parallel components are concealed [44, p.112].

A Characterisation of Meaningful Test Drivers for Reactive Systems The following theorem characterises the trace behaviour of tests used by meaningful test drivers for reactive systems. An interpretation of its intuitive meaning is presented after its proof.

Theorem 11 *Let $\mathcal{D}(X)$ be a test driver for the test against SPEC in a standard configuration $(E, ASYS, SYS, I)$ of reactive systems. Define*

$$D_0 = D_0(\langle\,\rangle)$$

$$D_0(s) = \sqcap_{R:Ref(E_I/s)}(x : ([E_I/s]^0 \setminus R) \rightarrow D_0(s, x))$$

$$\begin{aligned} D_0(s, x) = {} & \textbf{if } x \in [ASYS_I/s]^0 \\ & \quad \textbf{then } (D_0(s \frown \langle x\rangle) \sqcap w \rightarrow SKIP) \\ & \quad \textbf{else } E/s \frown \langle x\rangle \end{aligned}$$

Then

1. *If $\mathcal{D}(X)$ is meaningful for the test against SPEC in the sense of Definition 8, then every test case U appearing as a parameter of $\mathcal{D}(X)$ satisfies $D_0 \sqsubseteq_T (U \parallel E_I)$.*
2. *If $D_0 \sqsubseteq_T (U \parallel E_I)$, U is admissible and $s \frown \langle w\rangle \in Traces(U)$ implies $s \in Traces(E_I)$ for every test case U, then $\mathcal{D}(X)$ ist meaningful for the <u>may</u>-test against SPEC.*

Proof.
Proof Obligation 1: We show that $[D_0/s]^0 \setminus \{w\} = [E_I/s \upharpoonright I]^0$ for all traces s of D_0, so that, as a consequence, $D_0 \sqsubseteq_T E_I$.

The definitions of $D_0(s)$ and $D_0(s, x)$ show that for every s we have either $[D_0/s]^0 = \bigcup_{R:Ref(E_I/s\upharpoonright I)}([E_I/s\upharpoonright I]^0 \setminus R)$ or $[D_0/s]^0 = \{w\} \cup \bigcup_{R:Ref(E_I/s\upharpoonright I)}([E_I/s\upharpoonright I]^0 \setminus R)$. Since $\emptyset \in Ref(E_I/s)$ this yields $[D_0/s]^0 = [E_I/s \upharpoonright I]^0$ or $[D_0/s]^0 = \{w\} \cup [E_I/s \upharpoonright I]^0$, which proves Obligation 1.

Proof of Part 1. Let $\mathcal{D}(X)$ be meaningful for the test against *SPEC* and U be a test case occurring in $\mathcal{D}(X)$. We show that $D_0 \sqsubseteq_T (U \parallel E_I)$, which is is equivalent to $Traces(U \parallel E_I) \subseteq Traces(D_0)$:

Case 1.1. Let $s \in Traces(U \parallel E_I) \wedge \neg\, (\langle w\rangle \textbf{ in } s)$. Then the definition of the parallel operator implies that $s \in Traces(E_I)$. From Obligation 1 $s \in Traces(D_0)$ follows. This proves part 1 in Case 1.1.

Case 1.2. Let $s^\frown\langle w\rangle^\frown u \in \mathit{Traces}(U \parallel E_I)$. Since U is an admissible test case in the sense of Definition 3, $\neg\ (\langle w\rangle\ \mathbf{in}\ s^\frown u)$ and $u = \langle\ \rangle$ follows, so s contains I-events only, and $s \in \mathit{Traces}(D_0)$ follows as in Case 1.1. Since U is meaningful in the sense of Definition 8, we have $s \in \mathit{Traces}(\mathit{SPEC})$, because U indicates success after s. Inserting the reactive system structure for SPEC implies $s \in \mathit{Traces}(E_I \parallel \mathit{ASYS}_I)$. The $\parallel$-law implies $s \in \mathit{Traces}(\mathit{ASYS}_I)$. Then the definition of $D_0(\mathit{front}(s), \mathit{last}(s))$ implies $w \in [D_0/s]^0$. Therefore we have $s^\frown\langle w\rangle \in \mathit{Traces}(D_0)$. This shows Case 1.2 and completes the proof of part 1 of the theorem.

Proof of Part 2. Let $D_0 \sqsubseteq_T (U \parallel E_I)$ and $(\forall\, s \bullet s^\frown\langle w\rangle \in \mathit{Traces}(U) \Rightarrow s \in \mathit{Traces}(E_I))$ be valid for every test case U appearing as a parameter in $\mathcal{D}(X)$. We show that $\mathcal{D}(X)$ performs meaningful $\underline{\mathit{may}}$-test executions against SPEC:

Let $s^\frown\langle w\rangle \in \mathit{Traces}(U)$. We have to show that $s \in \mathit{Traces}(\mathit{SPEC})$. According to the assumptions, s is a trace both of U and E_I, therefore the law about $\parallel$ yields $s^\frown\langle w\rangle \in \mathit{Traces}(U \parallel E_I)$. Since $(U \parallel E_I)$ refines D_0 in the trace model, $s^\frown\langle w\rangle \in \mathit{Traces}(D_0)$ follows. According to the definition of $D_0(\mathit{front}(s), \mathit{last}(s))$, this implies $s \in \mathit{Traces}(\mathit{ASYS}_I)$. As a consequence, $s \in \mathit{Traces}(E_I) \cap \mathit{Traces}(\mathit{ASYS}_I)$, so since $\mathit{SPEC} = (E_I \parallel \mathit{ASYS}_I)$, this results in $s \in \mathit{Traces}(\mathit{SPEC})$. This completes the proof of part 2 of the theorem.
□

Interpretation of Theorem 11 Since any test case U in the reactive system configuration runs in parallel with E_I, it is ensured that only traces of E_I may be performed. Therefore there is no need for D_0 to admit other traces than those contained in $\mathit{Traces}(E_I)$. A meaningful test case will only indicate success, if the trace performed so far is also a valid trace according to the specification ASYS of the target system. The length of the trace performed before the indication of success depends on the test case U. Due to these initial considerations, process D_0 is designed as a simulation of the environment E as far as it is relevant for the investigation of correctness properties of the target system, i. e., E projected onto the events contained in I. D_0 performs a trace s of I-events, such that s is also a trace of E_I. At each execution step, D_0 chooses a refusal of E_I/s at random and then engages into any event that would not be refused by E_I in this situation. If the next event x is also valid according to the specification of the target system, i. e., $x \in [\mathit{ASYS}_I/s]^0$, D_0 will non-deterministically either continue with the next execution step or indicate success w. If x is not contained in $[\mathit{ASYS}_I/s]^0$, this represents a safety failure. After having indicated success, D_0 does not produce further events, in accordance with the fact that admissible test cases terminates after w. After a failure D_0 will admit any trace that might be performed by E_I. This is necessary to allow the test cases U compared against D_0 to terminate at an arbitrary (finite) trace after failure. Observe that D_0 itself is *not* an admissible test case, since an occurrence of w may be refused. As a consequence, the refinement relation $D_0 \sqsubseteq_T (U \parallel E_I)$ assumed for the second part of the theorem does not allow to conclude the admissibility of the test case U.

A Trustworthy $\sqsubseteq_{FD}$-Test Driver for Reactive Systems After the general characterisation of trustworthy test drivers presented above, our objective is now to introduce a specific test driver with the following properties:

- The test driver is trustworthy for $\sqsubseteq_{FD}$ -test, against associated standard configurations $(E, ASYS, SYS, I)$.
- The test driver is implementable in a "straightforward" way, if the specifications of the abstract target system $ASYS$ and of the environment process E are available in the transition system representation described in 4.5.1.
- The test driver simulates the environment process E, so that only the target system SYS must be provided for a test execution.

The test driver will use test cases derived from the Hennessy Test Cases according to Theorem 8.

Given a fixed standard configuration $(E, ASYS, SYS, I)$ of a reactive system, we will now introduce a test driver which is trustworthy for $\sqsubseteq_{FD}$ -test.

Theorem 12 *Let $(E, ASYS, SYS, I)$ be a standard configuration of a reactive system. Define a collection $\mathcal{U} = \{U(n) \mid n \in \mathbb{N}\}$ of test cases by:*

$$U(n) = U(n, \langle\, \rangle)$$

$$\begin{aligned}
U(n, s) = &\\
&(\#s = n \vee A(s) = \emptyset)\&(w \rightarrow SKIP)\\
&\Box\\
&(\#s < n)\&(e : ([E_I/s]^0 \setminus [ASYS_I/s]^0) \rightarrow \dagger \rightarrow SKIP)\\
&\Box\\
&(\#s < n-1 \wedge R(s) \neq \emptyset)\&(\sqcap_{R:R(s)} U(n, s, [(E \parallel ASYS)_I/s]^0 \setminus R))\\
&\Box\\
&(\#s = n-1 \wedge A(s) \neq \emptyset)\&(\sqcap_{R:refMax(E_I/s), A:A(s)} U(n, s, A \setminus R))
\end{aligned}$$

$$U(n, s, M) = e : M \rightarrow U(n, s ^\frown \langle e \rangle)$$

where

$$\begin{aligned}
A(s) = \{A : \mathbb{P}\, I \mid\ & A \subseteq [(E \parallel ASYS)_I/s]^0\\
& \wedge (\forall R : Ref((E \parallel ASYS)_I/s) \bullet A \not\subseteq R)\\
& \wedge (\forall X : \mathbb{P}\, A - \{A\} \bullet (\exists R : Ref((E \parallel ASYS)_I/s) \bullet X \subseteq R))\}
\end{aligned}$$

and

$$R(s) = \{R : Ref(E_I/s) \mid [(E \parallel ASYS)_I/s]^0 \setminus R \neq \emptyset\}.$$

Then

1. *If $Traces(E_I) \cap Div(ASYS_I) = \emptyset$ then $ASYS_I$ <u>must</u> $U(n)$ for all test cases $U(n) \in \mathcal{U}$.*
2. *The following statements are equivalent:*
   (a) *SYS_I <u>must</u> $U(n)$ for all test cases $U(n) \in \mathcal{U}$.*
   (b) *$Traces(E_I) \cap Div(SYS_I) = \emptyset$ and $(E \parallel ASYS)_I \sqsubseteq_{FD} (E \parallel SYS)_I$.*

$\Box$

Interpretation of Theorem 12 Each test case $U(n)$ explores the behaviour of the target system for traces s of length $\#s \leq n$. The basic idea of the structure of $U(n)$ is to simulate the environment E_I with respect to traces and refusals while exercising a combination of test cases $U_S(s,a)$ and $U_C(s,A)$ on the target system. $U(n)$ uncovers all trace failures up to length $n-1$ and detects all requirements coverage violations occurring in the next step after having run through a trace of length $n-1$. This means that the tasks of the test oracle are integrated in the test cases and performed during their execution. This is called *on-the-fly test evaluation.* $U(n,s)$ represents the state of a test execution where trace s has already been successfully performed. At each execution step, $U(n,s)$ will detect any event $e \in ([E_I/s]^0 \setminus [ASYS_I/s]^0)$, which is acceptable according to the environment but corresponds to a trace failure of the target system SYS. This will be indicated by a special event †, signalling failure of the test, if the target system does not diverge before indication becomes possible.

As long as $\#s < n-1$, $U(n,s)$ will behave as E_I/s with respect to the refusal of events: In the third $\Box$-branch an arbitrary refusal $R \in R(s)$ may be selected, and every event outside R that may be performed by $(E \parallel ASYS)_I/s$ is offered to the target system. $R(s)$ is the set of all E_I/s-refusals that do not block further operation of $(E \parallel ASYS)_I$ completely. If the target system SYS_I may "legally" block in environment E_I after trace s because the same holds for $ASYS_I$, this situation is reflected by $A(s) = \emptyset$. Now the first $\Box$-branch offers successful termination because such a deadlock may occur nondeterministically, but does not indicate failure. At the same time the third $\Box$-branch still offers events $[(E \parallel ASYS)_I/s]^0$ for further successful execution, so that full trace coverage can be reached if every possible execution of $(U(n) \parallel SYS_I)$ is carried out. Since $Ref(E_I/s)$ is subset-closed, $R(s)$ is empty iff $[(E \parallel ASYS)_I/s]^0 = \emptyset$, that is iff $(E \parallel ASYS)_I$ always deadlocks after s.

For $\#s = n$, $U(n,s)$ will only admit events contained in a minimum set $A \in A(s)$ that cannot be completely refused by $(E \parallel ASYS)_I/s$. Therefore $U(n)$ can detect requirement coverage failures of SYS occurring after traces of length n, when running in environment E. There is a subtle difference between the third and the fourth ALT-branch: To detect requirement coverage failures in the forth branch it obviously suffices to select the *maximum* refusals R in the expressions $A \setminus R$. If $(E \parallel ASYS)_I/s = P \sqcap STOP$ for some process P, the maximum refusal is the full alphabet I and $A(s)$ is empty, so there is nothing to investigate about non-blocking properties. In contrast to that it has to be ensured in the third $\Box$-branch that every possible continuation after s is inspected. Therefore also smaller refusals in $R(s)$ have to be selected, so that the possibility to enter the P-branch in process $P \sqcap STOP$ will be provided for the target system.

The internal choice operators ($\sqcap$) used in the definition of $U(n,s)$ show where internal decisions with respect to the control of the test executions may be taken: At each execution step $U(n,s)$ the refusals R or the sets A may be selected according to a test coverage strategy implemented in the test driver. Since there are many possibilities for suitable strategies, these are hidden in the definition of $U(n)$. Any strategy covering all possible executions of $U(n)$ is valid.

Using LTS representations for the CSP specifications of E_I and $ASYS_I$, test $U(n)$ is implementable in a straight forward way: $U(n)$ is determined by the traces and refusals of E_I and $ASYS_I$, and these are contained in the corresponding LTS representations.

Part 1 of Theorem 12 states that the test cases $U(n)$ are complete in the sense that they will always execute successfully when the target system and its abstract specification process $ASYS$ show *identical* behaviour at the system interface. This is only ensured when the abstract specification process $ASYS_I$ of the target system does not diverge in environment E. We do not consider this a severe restriction to the theorem's usability, since explicit incorporation of divergence into abstract specifications occurs very rarely in practice. The implication of part 2, (a) $\Rightarrow$ (b) states the soundness property that successful execution of the tests implies failures-divergence refinement when operating in environment E. Moreover, since the $U(n)$ never diverge by construction, the successful execution ensures that the target system will never diverge in this environment. Conversely, if the refinement relation has been already established and it is ensured that the target system will not diverge in its environment E, we can be certain that all tests $U(n)$ will succeed for SYS_I.

Before presenting the proof of the Theorem 12, we will introduce a lemma about Hennessy Tests of type $U_C(s, A)$ in the context of reactive systems.

Lemma 15 *Given* P, Q *with* $\alpha(P) = \alpha(Q)$, $s \in \alpha(P)^*$ *and* $A \subseteq \alpha(P)$ *with* $A \neq \emptyset$. *Then*

1. $(P \parallel Q)\ \underline{must}\ U_C(s, A) \wedge s \in \mathit{Traces}(P \parallel Q) \Rightarrow P\ \underline{must}\ U_C(s, A) \wedge Q\ \underline{must}\ U_C(s, A)$
2. $(P \parallel Q)\ \underline{must}\ U_C(s, A) \Rightarrow P\ \underline{must}\ U_C(s, A) \vee Q\ \underline{must}\ U_C(s, A)$

Proof.
Proof of 1. Suppose that $(P \parallel Q)\ \underline{must}\ U_C(s, A) \wedge s \in \mathit{Traces}(P \parallel Q)$ holds but $P\ \underline{must}\ U_C(s, A) \wedge Q\ \underline{must}\ U_C(s, A)$ is false. Specifically, assume that $\neg(P\ \underline{must}\ U_C(s, A))$ holds. Then $(s, A) \in \mathit{Fail}(P)$ by Lemma 14. Since $s \in \mathit{Traces}(P \parallel Q)$ and $\alpha(P) = \alpha(Q)$, also $s \in \mathit{Traces}(Q)$ holds. From to the semantic definition of $\parallel$, $(s, A) \in \mathit{Fail}(P \parallel Q)$ follows. Lemma 14 now implies $\neg((P \parallel Q)\ \underline{must}\ U_C(s, A))$, a contradiction. The same argument can be applied for Q. This proves 1.

Proof of 2. Applying contraposition, we assume that neither $P\ \underline{must}\ U_C(s, A)$ nor $Q\ \underline{must}\ U_C(s, A)$ hold. Then Lemma 14 implies $(s, A) \in \mathit{Fail}(P)$ and $(s, A) \in \mathit{Fail}(Q)$. Again, the semantic definition of $\parallel$ implies $(s, A) \in \mathit{Fail}(P \parallel Q)$, so $\neg((P \parallel Q)\ \underline{must}\ U_C(s, A))$ follows. This completes the proof of the lemma.
□

Example 4.9 Part 1 of Lemma 15 is really as strong as possible: Define $P = a \rightarrow b \rightarrow STOP$ and $Q = a \rightarrow c \rightarrow CHAOS$ with $\alpha(P) = \alpha(Q) = \{a, b, c\}$. Then, for arbitrary A, $P\ \underline{must}\ U_C(\langle a, c\rangle, A)$ and $\neg\ (Q\ \underline{must}\ U_C(\langle a, c\rangle, A))$, but $(P \parallel Q)\ \underline{must}\ U_C(\langle a, c\rangle, A)$.
□

Example 4.10 The converse of Part 2 of Lemma 15 is not true: Define $P = a \rightarrow b \rightarrow STOP$ and $Q = a \rightarrow c \rightarrow STOP$ with $\alpha(P) = \alpha(Q) = \{a, b, c\}$. Then $P\ \underline{must}\ U_C(\langle a\rangle, \{b, c\})$ and $Q\ \underline{must}\ U_C(\langle a\rangle, \{b, c\})$, but $\neg\ ((P \parallel Q)\ \underline{must}\ U_C(\langle a\rangle, \{b, c\}))$.
□

Proof of Theorem 12: We will first state and verify four proof obligations that are used to establish the validity of Theorem 12, 2.(a) $\Rightarrow$ 2.(b).

Proof Obligation 1. Show that

$$s \in \mathit{Traces}((E \,\|\, ASYS)_I) \wedge \#s < n \Rightarrow s \in \mathit{Traces}(U(n)) \wedge U(n)/s = U(n,s)$$

Given $U(n)$ with fixed $n > 0$, we will use induction over the length $m = \#s$ of the traces s, $m = 0, \ldots, n-1$. The assertion holds trivially for $m = 0$, since this implies $s = \langle\,\rangle$ and $U(n)/\langle\,\rangle = U(n) = U(n, \langle\,\rangle)$ by definition. Assume that obligation 1 has been proven for $m \geq 0$ and let $s \in \mathit{Traces}((E \,\|\, ASYS)_I) \wedge \#s = m+1 \wedge m+1 < n$. Define trace $u = \mathit{front}(s)$ and event $a = \mathit{last}(s)$.

Applying the induction hypothesis to u yields $u \in \mathit{Traces}(U(n))$ and $U(n)/u = U(n,u)$. Since $\#s < n$ we have $\#u < n-1$. Furthermore, $s \in \mathit{Traces}((E \,\|\, ASYS)_I)$ implies $a \in [(E \,\|\, ASYS)_I/u]^0$, so $R(u) \neq \emptyset$. Evaluating the guards in the definition of $U(n,s)$ therefore yields

$$\begin{aligned} U(n)/u = U(n,u) = \;& (A(u) = \emptyset)\&(w \to SKIP) \\ & \Box \\ & (e : ([E_I/u]^0 \setminus [ASYS_I/u]^0) \to \dagger \to SKIP) \\ & \Box \\ & (\sqcap_{R:R(u)} U(n,u,[(E \,\|\, ASYS)_I/u]^0 \setminus R)) \end{aligned}$$

$$U(n,u,M) = e : M \to U(n, u^\frown\langle e\rangle)$$

Since $\emptyset \in R(u)$, process $U(n,u,[(E \,\|\, ASYS)_I/u]^0)$ is an alternative of the $(\sqcap_{R:R(u)} \ldots)$-branch in $U(n,u)$. As a consequence, $[U(n,u,[(E \,\|\, ASYS)_I/u]^0)]^0 \subseteq [U(n)/u]^0$ holds. By construction of the processes $U(n,s,M)$ we have

$$[U(n,u,[(E \,\|\, ASYS)_I/u]^0)]^0 = [(E \,\|\, ASYS)_I/u]^0$$

Since $a \in [(E \,\|\, ASYS)_I/u]^0$ we get $a \in [U(n)/u]^0$. This proves $s = u^\frown\langle a\rangle \in \mathit{Traces}(U(n))$.

Since the first and the second $\Box$-branch of $U(n)/u$ do not accept a, we obtain $U(n)/u^\frown\langle a\rangle = U(n,u,[(E \,\|\, ASYS)_I/u]^0 \setminus R)/\langle a\rangle$ for some $R \in R(u)$ with $a \in [(E \,\|\, ASYS)_I/u]^0 \setminus R$. (Note, that for all such sets R, R' we have by definition of $U(n,s,M)$ that $U(n,u,[(E \,\|\, ASYS)_I/u]^0 \setminus R)/\langle a\rangle = U(n,u,[(E \,\|\, ASYS)_I/u]^0 \setminus R')/\langle a\rangle)$. Hence, we conclude

$$\begin{aligned} U(n)/s = U(n)/u^\frown\langle a\rangle &= U(n,u,[(E \,\|\, ASYS)_I/u]^0 \setminus R)/\langle a\rangle \\ &= U(n, u^\frown\langle a\rangle) = U(n,s) \end{aligned}$$

which proves obligation 1.

Proof Obligation 2. For $U_S(s,a) \in \mathcal{H}_S((E \,\|\, ASYS)_I)$ and $\#s = n-1$ with $\mathcal{H}_S$ defined as in Theorem 8 show that

$$\neg\, ((E \,\|\, SYS)_I \;\underline{must}\; U_S(s,a)) \Rightarrow \neg\, (SYS_I \;\underline{must}\; U(n)).$$

By Lemma 14, $\neg\, ((E \,\|\, SYS)_I \;\underline{must}\; U_S(s,a))$ implies $s^\frown\langle a\rangle \in \mathit{Traces}((E \,\|\, SYS)_I)$. From the semantic definition of $\|$, $s^\frown\langle a\rangle \in \mathit{Traces}(E_I)$ and $s^\frown\langle a\rangle \in \mathit{Traces}(SYS_I)$ follows. Since

$U_S(s,a) \in \mathcal{H}_S$, $s \in \mathit{Traces}((E \,\|\, \mathit{ASYS})_I) \wedge a \notin [(E \,\|\, \mathit{ASYS})_I/s]^0$, so the semantics of $\|$ yields $s \in \mathit{Traces}(\mathit{ASYS}_I) \wedge s\frown\langle a\rangle \notin \mathit{Traces}(\mathit{ASYS}_I)$. As a consequence, $a \in [E_I/s]^0 \setminus [\mathit{ASYS}_I/s]^0$. From the validity of Proof Obligation 1 we get $U(n)/s = U(n,s)$, so $U(n)/s\frown\langle a\rangle = U(n,s)/\langle a\rangle$ will accept the branch $\dagger \to \mathit{SKIP}$, because $\#s < n$ and $a \in [E_I/s]^0 \setminus [\mathit{ASYS}_I/s]^0$. This means, that at least in one execution $(U(n) \,\|\, \mathit{SYS})$ will fail after $s\frown\langle a\rangle$, which shows the validity of Proof Obligation 2.

Proof Obligation 3. For $U_C(s,A) \in \mathcal{H}_C((E \,\|\, \mathit{ASYS})_I)$ and $\#s = n-1$ show that

$$\neg\,((E \,\|\, \mathit{SYS})_I \ \underline{must}\ U_C(s,A)) \Rightarrow \neg\,(\mathit{SYS}_I \ \underline{must}\ U(n))$$

$U_C(s,A) \in \mathcal{H}_C((E \,\|\, \mathit{ASYS})_I)$ and part 1 of Lemma 15 imply $A \in A(s) \wedge s \in \mathit{Traces}((E \,\|\, \mathit{ASYS})_I) \wedge E_I \ \underline{must}\ U_C(s,A) \wedge \mathit{ASYS}_I \ \underline{must}\ U_C(s,A)$. Since $\neg\,((E \,\|\, \mathit{SYS})_I \ \underline{must}\ U_C(s,A))$, Lemma 14 implies $(s,A) \in \mathit{Fail}((E \,\|\, \mathit{SYS})_I)$, that is,

$$(\exists\, R : \mathit{refMax}((E_I \,\|\, \mathit{SYS}_I)/s) \bullet A \cap [(E_I \,\|\, \mathit{SYS}_I)/s]^0 \subseteq R).$$

The definition of $\|$ implies that such a refusal R is a union of maximum refusals $X_1 \in \mathit{refMax}(E_I/s)$ and $X_2 \in \mathit{refMax}(\mathit{SYS}_I/s)$, so $A \cap [E_I/s]^0 \cap [\mathit{SYS}_I/s]^0 \subseteq (X_1 \cup X_2)$. $E_I \ \underline{must}\ U_C(s,A)$ implies $A \setminus X_1 \neq \emptyset$. Moreover, $A \subseteq [E_I/s]^0$ holds according to the definition of $\mathcal{H}_C((E \,\|\, \mathit{ASYS})_I)$, so $(A \setminus X_1) \cap [\mathit{SYS}_I/s]^0 \subseteq X_2$. This means that SYS_I/s may refuse every event of $A \setminus X_1$.

From the validity of Proof Obligation 1 we know that $U(n)/s = U(n,s)$. Evaluation of the guards in the definition of $U(n,s)$ results in:

$$\begin{aligned} U(n)/s = U(n,s) = \ & (e : ([E_I/s]^0 \setminus [\mathit{ASYS}_I/s]^0) \to \dagger \to \mathit{SKIP}) \\ & \square \\ & (\sqcap_{R' : \mathit{refMax}(E_I/s),\, A' : A(s)} U(n,s,A' \setminus R')) \end{aligned}$$

$$U(n,s,M) = e : M \to U(n, s\frown\langle e\rangle)$$

Therefore, since $A \in A(s)$ and $X_1 \in \mathit{refMax}(E_I/s)$, a possible behaviour of $U(n)/s$ is defined by the process

$$\begin{aligned} P = \ & (e : ([E_I/s]^0 \setminus [\mathit{ASYS}_I/s]^0) \to \dagger \to \mathit{SKIP}) \\ & \square \\ & (e : A \setminus X_1 \to U(n, s\frown\langle e\rangle)). \end{aligned}$$

At least one execution of $((E \,\|\, \mathit{SYS})_I/s \,\|\, P)$ will fail: If SYS_I/s refuses X_2, the second P-branch is blocked, and the first branch leads to trace failure $\dagger$. This proves obligation 3.

Proof Obligation 4. For $U_D(s) \in \mathcal{H}_D((E \,\|\, \mathit{ASYS})_I)$ and $\#s = n-1$ show that

$$\neg\,((E \,\|\, \mathit{SYS})_I \ \underline{must}\ U_D(s)) \Rightarrow \neg\,(\mathit{SYS}_I \ \underline{must}\ U(n))$$

The premise $U_D(s) \in \mathcal{H}_D((E \,\|\, \mathit{ASYS})_I)$ implies $s \in \mathit{Traces}(E_I \,\|\, \mathit{ASYS}_I)$ and $s \notin \mathit{Div}(E_I \,\|\, \mathit{ASYS}_I)$. The semantics of $\|$ implies $s \notin \mathit{Div}(E_I) \wedge s \notin \mathit{Div}(\mathit{ASYS}_I)$. Let

$u \leq s$ such that $u \in Div((E \,\|\, SYS)_I)$. Since u is a prefix of s, $u \notin Div(E_I)$ also holds and $u \in Div(SYS_I)$ follows. Since $s \in Traces((E \,\|\, ASYS)_I)$, the validity of proof obligation 1 shows that $u, s \in Traces(U(n))$. Therefore $u \in Traces(U(n) \,\|\, SYS_I)$, and the definition of $\|$ yields $u \in Div(U(n) \,\|\, SYS_I)$. As a consequence, $(U(n) \,\|\, SYS_I)$ may refuse any event of $I \cup \{w\}$ after having engaged into trace u. Since u does not contain w, such an execution will fail. This proves Obligation 4.

Proof of Theorem 12, 2.(a) $\Rightarrow$ 2.(b). Suppose $SYS_I \; \underline{must} \; U(n)$ for all $n \in \mathbb{N}$ and let $U_D(s) \in \mathcal{H}_D((E \,\|\, ASYS)_I)$. Then $(E \,\|\, SYS)_I \; \underline{must} \; U_D(s)$, because otherwise $\neg \, (SYS_I \; \underline{must} \; U(\#s+1))$ according to proof obligation 4. Let $U_S(s, a) \in \mathcal{H}_S((E \,\|\, ASYS)_I)$. Then $(E \,\|\, SYS)_I \; \underline{must} \; U_S(s, a)$, because otherwise $\neg \, (SYS_I \; \underline{must} \; U(\#s + 1))$ according to proof obligation 2. Let $U_C(s, A) \in \mathcal{H}_C((E \,\|\, ASYS)_I)$. Then $(E \,\|\, SYS)_I \; \underline{must} \; U_C(s, A)$, because otherwise $\neg \, (SYS_I \; \underline{must} \; U(\#s + 1))$ according to proof obligation 3.

Now we have established that $(E \,\|\, SYS)_I \; \underline{must} \; U$ holds for all $U \in \mathcal{H}_S((E \,\|\, ASYS)_I) \cup \mathcal{H}_C((E \,\|\, ASYS)_I) \cup \mathcal{H}_D((E \,\|\, ASYS)_I)$. Therefore the application of Theorem 8 results in $(E \,\|\, ASYS)_I \sqsubseteq_{FD} (E \,\|\, SYS)_I$. This proves Theorem 12, 2.(a) $\Rightarrow$ 2.(b).

Proof of Theorem 12, 2.(b) $\Rightarrow$ 2.(a). We apply contraposition and prove that the negation of the premise 2.(a) implies the negation of 2.(b). Suppose $\neg \, (SYS_I \; \underline{must} \; U(n))$ for some $n \in \mathbb{N}$. Analysis of the structure of $U(n)$ shows that an execution of $(U(n) \,\|\, SYS_I)$ can only fail iff at least one of the following conditions are true:

1. An execution diverges, before success could be signalled:
   $(\exists \, s : Traces(U(n) \,\|\, SYS_I) \bullet \neg \, \langle w \rangle \textbf{ in } s \wedge s \in Div(U(n) \,\|\, SYS_I)$.
2. An execution produces a trace failure:
   $(\exists \, s : Traces(U(n) \,\|\, SYS_I) \bullet \#front(s) < n \wedge last(s) \in [E_I/s]^0 \setminus [ASYS_I/s]^0)$.
3. An execution blocks in the third $\square$-branch of $U(n, s)$:
   $(\exists \, s : Traces(U(n) \,\|\, SYS_I);\; R, X : \mathbb{P}\, I \bullet \#s < n - 1 \wedge (s, X) \in Fail(SYS_I) \wedge A(s) \neq \emptyset \wedge R \in R(s) \wedge [(E \,\|\, ASYS)_I/s]^0 \setminus R \cap [SYS_I/s]^0 \setminus X = \emptyset)$.
4. An execution blocks in the forth $\square$-branch of $U(n, s)$:
   $(\exists \, s : Traces(U(n) \,\|\, SYS_I);\; A, R, X : \mathbb{P}\, I \bullet \#s = n - 1 \wedge (s, X) \in Fail(SYS_I) \wedge A \in A(s) \wedge (s, R) \in Fail(E_I) \wedge A \setminus (R \cup X) = \emptyset)$.

Observe that, due to the specification of $U(n)$, s is also contained in $Traces(E_I)$ in all four cases. Moreover, for cases 3. and 4. s is also contained in $Traces((E \,\|\, ASYS)_I)$.

Since $Div(U(n)) = \emptyset$ by construction, failure condition 1 implies $s \in Div(SYS_I)$. As a consequence, $Traces(E_I) \cap Div(SYS_I) \neq \emptyset$, so the negation of 2.(b) holds.

For failure condition 2, we have that $front(s) \in Traces((E \,\|\, ASYS)_I)$, but $last(s) \notin [(E \,\|\, ASYS)_I/front(s)]^0$. Then the test case $U_S(front(s), last(s))$ is contained in $\mathcal{H}_S((E \,\|\, ASYS)_I)$, and this test fails for the execution of $(E \,\|\, SYS)_I$ when trace s is produced. Again this violates 2.(b), because $(E \,\|\, ASYS)_I \sqsubseteq_{FD} (E \,\|\, SYS)_I$ implies $(E \,\|\, SYS)_I \; \underline{must} \; U_S(front(s), last(s))$ according to Theorem 8.

For failure condition 3, there exists an $A \in A(s)$ such that $(E \,\|\, ASYS)_I \; \underline{must} \; U_C(s, A)$, so $U_C(s, A)$ is an element of $\mathcal{H}_C((E \,\|\, ASYS)_I)$. This test will fail for at least one execution of

$(U_C(s, A) \,\|\, (E \,\|\, SYS)_I)$, since this parallel composition may block completely after s. This implies that $(E \,\|\, SYS)_I$ does not refine $(E \,\|\, ASYS)_I$ due to a requirements coverage failure.

For failure condition 4, observe that $(s, R \cup X)$ is a failure of $(E \,\|\, SYS)_I$, so at least one execution of $(U_C(s, A) \,\|\, (E \,\|\, SYS)_I)$ will fail. However, $(E \,\|\, ASYS)_I$ $\underline{must}$ $U_C(s, A)$ holds since $A \in A(s)$. Therefore $U_C(s, A)$ is contained in $\mathcal{H}_C((E \,\|\, ASYS)_I)$. Again, this implies that $(E \,\|\, SYS)_I$ does not refine $(E \,\|\, ASYS)_I$ due to a requirements coverage failure. Now we have established that the negation of 2.(b) holds for each of the failure conditions listed above. This proves Theorem 12, 2.(b) $\Rightarrow$ 2.(a).

Proof of Theorem 12, 1. Choosing $SYS_I = ASYS_I$ and noting that $\sqsubseteq_{FD}$ is reflexive, part 1. of the theorem is a trivial consequence of part 2., (b) $\Rightarrow$ (a). This completes the proof of Theorem 12.
□

Using the results of Theorem 10 and Theorem 12, now we can state that test drivers using the test cases $U(n)$ have the desired correctness properties:

Theorem 13 *For a given standard configuration $(E, ASYS, SYS, I)$ of a reactive system, let the associated tests $U(n)$ be defined as above. Then the test driver*

$$\begin{aligned}\mathcal{D}(X) = (n := 0);\ *(U(n) \,\|\, X\hat{\ }(w \rightarrow & \mathit{monitor}?\mathit{next} \\ & \rightarrow (\textbf{if}\ \mathit{next}\ \textbf{then}\ i := i + 1;\ \mathit{SKIP}\ \textbf{else}\ \mathit{SKIP})));\end{aligned}$$

is trustworthy for $\sqsubseteq_{FD}$ -test.

Proof.
In analogy to Theorem 10, $\mathcal{D}(X)$ applies test cases ordered by the length of the traces. Since by Theorem 12 $U(n)$ has the same capabilities to detect failures as the Hennessy test cases $U_S(s, a)$, $U_C(s, A)$ with $\#s \leq n$, Theorem 10 implies that $\mathcal{D}(X)$ is trustworthy for $\sqsubseteq_{FD}$ -test.
□

4.6 Discussion and Future Work

This report focussed on the development of test drivers performing automated generation, execution and evaluation of tests for reactive systems against CSP specifications. Given a correctness relation between specifications and implementations, a test driver should be capable of

- generating test cases for every possible correctness violation,
- exercising test cases on the target system, at the same time simulating proper environment behaviour,
- detecting every violation of the correctness requirements during test execution.

To obtain test drivers which are *provably correct* with respect to these objectives, we analysed Hennessy's testing theory in the framework of untimed CSP. Hennessy's test classes are suitable for the detection of safety failures, insufficient requirements coverage, divergence failures and insufficient robustness in an implementation and characterise the corresponding refinement notions. As a result of this analysis we characterised minimal subsets of Hennessy's test classes that are still sufficient for the detection of safety failures and insufficient requirements coverage. Furthermore we presented the top-level specification of a test driver as implemented in the VVT-RT system. It was demonstrated that a test driver implementing this specification possesses the three capabilities listed above, with respect to testing safety and requirements coverage.

The work presented in this report reflects a "building block" of a joint enterprise of ELPRO LET GmbH, JP Software-Consulting, Bremen University and Kiel University in the field of test automation for reactive real-time systems. The main activities focus on the following research and development topics, covering both theoretical investigations and implementation in the VVT-RT tool:

Tool Verification Because of the dependability required of a tool allowing to perform large parts of a test suite without human interaction, it is intended to verify the critical parts of the tools and obtain tool certifications permitting the application of VVT-RT for the test of safety-critical railway control and avionic systems. The formal test driver specification and its verification presented in this report are a starting point of these activities.

Real-Time Testing for Reactive Systems The approach presented in this report exploits properties of untimed CSP specifications and their representations as transition graphs. This concept is presently extended to a subset of timed CSP in the semantics described by Davies [18], see [89]. It allows to generate test drivers detecting timed safety violations and simulating environment properties in real-time, in addition to the properties described in this report.

Test Monitors and Test Coverage Analysis A crucial problem in the context of hardware-in-the-loop testing is presented by the fact that only the transition graph of the specification is available, while the graph of the implementation is unknown, because in general it will be impossible to provide a formal specification covering the complete software, firmware and hardware behaviour[9]. The task of a test monitor is to reconstruct the unknown transition graph of the implementation by means of the events observed at the black-box interface and additional monitoring channels. Since the creation of additional interfaces "revealing" internal states of the target system is a complex design and development activity, it is of great practical value to know the minimum amount of internal information sufficient to determine the test coverage achieved.

Test of Data Transformations A wide-spread heuristic approach to improve our understanding of large-scale systems is to use different modelling techniques for the description of the three fundamental aspects *transformational behaviour*, *dynamic system behaviour* and *data structures*. On a more formal level, this approach is reflected by the effort to combine formal methods focusing on the description of dynamic aspects with methods that are most suitable for the description and analysis of transformational behaviour and data structures.

[9] This situation is different from pure software tests, where the implementation is represented by software code, and therefore the associated transition graph may be – at least theoretically – constructed.

In [101], for example, Roscoe, Woodcock and Wulf describe a concept to specify data structures and sequential system components by means of Z [108], while using untimed CSP specifications for the description of causal relations between interacting sequential processes. This separation between data transformation and control is also a promising approach in the field of test automation. Indeed, the concept described in this report may be regarded as complementary to the efforts of Hörcher [83, 41] and Mikk [63] regarding test automation based on Z specifications: While Hörcher's and Mikk's method is successful in the field of automated test evaluation for data transformations specified by operational Z schemas, operating on data structures represented by Z state schemas, it does not provide a means to test the (timed or untimed) dynamic behaviour of the (possibly parallel) interaction of the isolated operations. Conversely, our present approach focuses on the test of dynamic behavioural aspects, but does not allow to test operations on complex mathematical data structures, as provided for example by the Z mathematical toolkit [108]. In the next stage of our research activity, we will therefore investigate an integration of these two test automation approaches, following the suggestions of Roscoe, Woodcock and Wulf about the combined use of Z and CSP.

Investigation of Test Strategies and Combination with Formal Verification In order to manage the complexity of large-scale systems, test and verification activities are usually performed on different levels, ranging from isolated sequential units to the full system. In particular, different techniques, like test and formal verification may be applied to different system components. The objective of such test and verification strategies is to reduce the amount of test coverage to be achieved on a specific level by relying on the results previously achieved on the lower levels *(Integration Testing)*. However, the trustworthiness of integration test strategies depends on the architecture of the system, describing the mode of interaction between the isolated components. Based on the compositional proof theory of CSP, it is possible to derive conditions regarding the system architecture that are sufficient for a given test strategy to be trustworthy.

Exercising test strategies on units, sub-systems and the full target system will allow to use different methods and tools on the test levels where they can be most efficiently applied. For example, the partition testing techniques as described by Grochtmann and Grimm [34] are most useful on unit level. Test of isolated sequential processes can be designed and evaluated by means of the methods based on Z, and the tests concerning proper dynamic behaviour by the methods described in this report.

Investigation of other Semantics Interleaving semantics as used for CSP and other process algebras are not always the best formal framework for the generation of tests: To achieve full requirements coverage with respect to specifications interpreted in an interleaving semantics, test cases have to be executed for *every* possible sequence of unsynchronised events. In many applications it is known *a priori* that one representative of a set of traces, consisting of different permutations of the same events, would be sufficient to be tested. These aspects might be better reflected by other types of semantics, as, for example, partial order semantics.

Hybrid Systems A future research activity will consider the test of hybrid systems, admitting analog interfaces and specifications involving continuous or differentiable functions.

5. Security Aspects: Trustworthy Evaluation of IT Security Products

5.1 Overview

Information technology (IT) security focuses on the *confidentiality*, *integrity* and *availability* of IT systems. Because insufficient or incorrectly implemented security functionality may cause severe damage in today's applications, standards have been created describing requirements to be fulfilled by trustworthy IT security products or systems [47, 111, 28, 29]. In Europe, the *Information Technology Security Evaluation Criteria (ITSEC)* [47] represent the most important standard. They describe an evaluation and certification procedure allowing to award certain *quality levels* (E1 [lowest], ... ,E6 [highest level]) to IT security products and to classify the strength of their security mechanisms. In addition to quality levels, *functionality classes* are defined ([47, Annex A]) describing different types of security functions which should be present in specific types of products. For the highest level E6 of evaluation quality (cf. [47, pp. 98]), it is required to provide a formal security model, a formal specification of the critical parts of security-enforcing functions and a formal architectural design. Moreover, it must be explained using both informal and formal verification techniques that the design meets the specifications of the security model. The requirements of the ITSEC are consistent with the *V-model* [112] discussed in Chapter 2, but they are specialised on IT security products and focus less on product development than on product evaluation.

In this chapter methods for the evaluation of IT security products according to the ITSEC are investigated. We describe an effort to improve the objectivity, quality and efficiency of the evaluation process by application of formal methods. To this end, a concept for the formal specification of certain ITSEC requirements is presented. Based on such formal specifications, it is possible not only to justify informally, but also to *prove* the consistency of the (formal) specification of a concrete product with the ITSEC requirements. We believe that application of such a technique should become mandatory for the evaluation of maximum-quality products according to the highest evaluation level E6. A characteristic feature of the approach to formalise ITSEC requirements is that it makes use of *generic* specification techniques allowing the formalised requirements to be applied to a wide spectrum of system types.

Our concept is illustrated by means of an example formalising a requirement of ITSEC functionality class F-C1. The *access control list* mechanism implemented to control file access in extended UNIX versions serves as an example of IT security product functionality. We describe the formal verification process to be applied in order to prove that this access control function fulfills the ITSEC requirement. The presentation given in this chapter is an extended version of the earlier paper [77].

5.2 Product Evaluation in Industry – Practical Experiences

In 1994, my department at DST received the accreditation as an evaluation laboratory for IT security products. In Germany this accreditation is awarded by the *Bundesamt für Sicherheit in der Informationstechnik (BSI)*, the authorities responsible for the certification of IT security products.

The concepts described in this chapter are based on experiences with IT security product evaluations performed at DST to qualify for the accreditation. These experiences are consistent with the criticism expressed frequently by other evaluation laboratories and companies developing and selling IT security products. The problems seem to be present both in Europe and the United States of America, where comparable evaluation procedures exist, based on the US standards [111, 28, 29]. The crucial points of this criticism may be summarised as follows:

- "Low-cost" evaluation procedures cannot be considered as trustworthy. Since the evaluation is performed by means of informal documentation reviews and only a very modest amount of testing is required, the evaluation results will depend too much on the intuition of the evaluators and therefore cannot really increase confidence in the product.
- Even for the lowest standards E1 and E2 product manufacturers regard the evaluation procedures as too time-consuming: Too often a product version had been outdated before the evaluation suite was finished!
- The high-quality evaluation procedures involving Formal Methods may lead to more trustworthy and more objective evaluation results. However, the costs for such an evaluation are estimated to be unacceptably high for the product developers. Though at least parts of IT security products have been developed and verified using formal methods (see [61, 116]), their number is still very small. In Germany for example, no product has ever been evaluated according to the highest quality standards E5 or E6 defined in the ITSEC[1].

A closer analysis performed at DST led to the following assessment of the main causes for this criticism:

1. **Natural-Language Style of the ITSEC** The ITSEC standard is written in natural-language style. This results in several ambiguities regarding the question how the ITSEC requirements should be "mapped" onto the product. As a consequence, the objectivity of the evaluation process cannot be guaranteed, because different evaluators and certification authorities might interpret the evaluation standard in different ways.
2. **Insufficient Observation of Functionality Classes** Evaluation according to the ITSEC requirements focuses very strongly on the quality levels E1,. . . ,E6, while putting less emphasis on the functionality of the product: It is not mandatory during the

[1] A project for the development of a high-quality security interface for a message handling system had been planned by the German Ministry of Defense for several years. Initially, its objective was to reach an E6 quality standard, but recently the security requirements have been re-defined to make use of standard software to be evaluated by lower evaluation criteria.

evaluation process to relate the product to one of the ITSEC functionality classes[2]. As a consequence, some certificates award a high quality level to products implementing very weak security functions. Obviously, this does not fulfill the users' needs, who will be most interested in a product rating helping them to decide whether the product *functionality* will meet their expectations[3].

3. **Non-Availability of Standard Security Architectures** It is extremely time consuming to verify the absence of security threats caused by system architecture (e. g. *covert channels* [19]). This is mainly due to the fact that these threats are invisible on specification level. The ITSEC do not describe any reference architectures which help to avoid such security gaps.

4. **Insufficient Observation of "Sophisticated" Security Threats** While being rather explicit about certain basic features of product functionality (e. g. *identification, authentication* or *access control*), the ITSEC requirements cover security threats caused by the "legal" use of function *combinations* which may violate system security only in an insufficient way. For example, the problems of *interference between users* (see [32, 33, 48, 101]) and of *inference channels* (see [109]) allowing to deduce classified data from unclassified knowledge are not described in the ITSEC. As a consequence, evaluation may fail to notice important security deficiencies.

5. **Missing Conformance Tests** In contrast to standards in other fields (e. g. telecommunications, compiler development) the informal requirements specification style used in the ITSEC does not allow to define conformance test suites which could be easily adapted for each product of a certain functionality class. Therefore the *penetration tests* required by the ITSEC [47, 3.37] have to be designed from scratch for each new IT security product evaluation.

6. **Insufficient Re-use of Evaluation Results** The criticism listed above leads to the fact that today's evaluation techniques and results cannot be re-used in a systematic way, because there is no systematic classification of IT security products. Therefore in many cases, the evaluation process will be too time-consuming and expensive. As a consequence the E6-evaluation and certification of maximum-quality products still appears to be wishful thinking, because nobody will invest into formal product verification if the verification results cannot be re-used for similar product variants.

This list of deficiencies motivated the effort explained in the following sections. It is driven by the idea that evaluation procedures should be formalised and at the same time made re-usable by means of *generic formal specifications* of standard functionality required for IT security products. In the example to follow we will focus on methods to overcome the problems 1 and 2 described above. Further possibilities are discussed in Section 5.6.

[2] This is motivated by the fact that due to the rapid changes in today's information technology, not every IT security product can be easily classified to fit into a pre-defined category.

[3] . . . while naively taking the product *quality* for granted.

5.3 A Formal Evaluation Approach

To overcome the above-mentioned problems regarding the evaluation of IT security products we suggest the following approach:

Functionality Classes as Generic Formal Product Specifications The collection of ITSEC functionality classes should be modified in the following directions: First, new classes should be added to cover additional types of IT security products. Second, each class should be extended by additional requirements regarding the missing security threats mentioned above. Third, each functionality class should be formulated as a *generic product specification*, so that a concrete IT security product can be viewed as an instantiation of its corresponding functionality class. This generic (requirements) specification should make use of *formal specification techniques*, so that the correspondence between the product and its generic specification can be treated *mathematically*, as soon as the product itself is described formally.

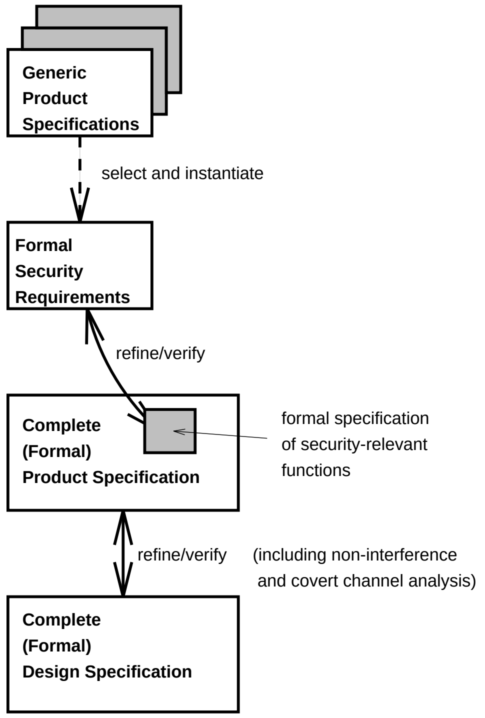

Figure 5.1: IT security evaluation approach.

Formal Product Evaluation Steps With a generic formal product specification at hand, an evaluation according to the highest quality level E6 could proceed along the following lines[4] (see Figure 5.1):

1. Associate the product with a suitable ITSEC functionality class.

[4]In general, the product developer will perform these steps; the evaluator will only check their correctness. However, it is also possible that the formalisation and verification process is completely performed in the evaluation laboratory; especially if the developer requires a high-quality evaluation but does not have the expertise to produce the formal specification and verification documents by himself

2. Instantiate this class with the concrete structures of the product. This will result in a new formal specification, which is free of any generic data types. This document can be viewed as a *formal security requirements specification* for the product: It does not describe the complete functionality of the product, but specifies the security requirements to be fulfilled by the product in order to comply with the functionality class.
3. Provide a formal product specification containing concrete specifications for all security-relevant operations.
4. Specify the abstraction relation between the security requirements specification obtained in step 2 and the formal product specification of step 3. This step will use concepts of operational and data refinement, as shown in Section 5.5.4.
5. Using the abstraction relation, prove that the concrete product uses data structures which are sufficient to implement the security requirements.
6. Prove that each security mechanism contained in the product is a correct implementation of the corresponding abstract operation. This step will use functional refinement proof techniques, as illustrated in Section 5.5.5. It will focus on each function separately.
7. Provide a formal design specification.
8. Prove that the product design meets the concrete formal specification. This refinement proof will have to consider the complete (in general parallel) system, because it has to be verified that the implemented security mechanisms cooperate in a way that does not violate the overall security of the product. It is well known that for this step it does not suffice to apply general refinement techniques for parallel systems (see for example [44]), but that additional design restrictions have to be considered to ensure that the refinement really preserves all the security properties specified (cf. [32, 33, 48, 101]).
9. Analyse the specification in combination with the design to prove that no unintended inference channels exist (cf. [109]).

In the subsequent sections we will illustrate the construction of generic formal product specifications and the evaluation steps 1,...,6.

5.4 Example: Security Evaluation Based on Generic Formal Specifications

To explain the application of formal methods as motivated in Section 1, we will now sketch the formalisation of a requirement defined for the ITSEC functionality class F-C1 which is the initial class of a sequence of increasingly strict security functionality requirements.

5.4.1 Functionality Class F-C1: Discretionary Access Control

In [47, A.9] *discretionary access control* is described as

> *"The TOE shall be able to distinguish and administer access rights between each user and the objects which are subject to the administration of rights, on the basis of an individual user, or on the basis of membership of a group of users, or both. It shall be possible to completely deny users or user groups access to an object."*

Here 'TOE' means *Target of Evaluation*, i. e. the product to be evaluated. The functional requirements of class F-C1 have been copied for the European ITSEC from the TCSEC [111]. Similar requirements can be found in [28, p. 63].

5.4.2 Formalisation of the F-C1 Requirement

The goal of this section is to formalise the above requirement, in order to make it unambiguous and to provide a basis for the application of formal verification techniques during product evaluation. As a formal description technique, we use the formal specification language Z, as defined in [108] and [117]. To make this chapter sufficiently self-contained, the language elements used will be explained while introducing the example. The formalisation contains certain decisions how the informal text quoted above should be interpreted. These decisions will be discussed below in Section 5.4.3 and 5.6.

The F-C1 requirement cited above references *objects, users* and *groups of users.* Each object combined with its access information may be characterised by the following *generic Z schema*:

$$
\begin{array}{|l}
\hline
\textit{Object}[\textit{OBJECTS}, \textit{USERS}, \textit{GROUPS}] \\
o : \textit{OBJECTS} \\
u : \mathbb{F}\ \textit{USERS} \\
\textit{uDenied} : \mathbb{F}\ \textit{USERS} \\
g : \mathbb{F}\ \textit{GROUPS} \\
\hline
u \cap \textit{uDenied} = \emptyset \\
\hline
\end{array}
$$

Here *OBJECTS, USERS, GROUPS* are unspecified generic data types which may be instantiated with the concrete structures of a product. $o, u, \textit{uDenied}, g$ are variable declarations with the following interpretation:

- o is the "logical contents" of the object.
- u is the finite set of users allowed to access the object.
- *uDenied* is the finite set of users explicitly excluded from object access.
- g is the finite set of groups, whose members are allowed to access the object.

The *predicate part* $u \cap \textit{uDenied} = \emptyset$ of the schema states the invariant condition that any user being allowed to access the object must not be registered at the same time in the set of users without object access.

When attempting to access an object, users have to exhibit their identification, denoted by $u? : \textit{USERS}$ and the groups $g? : \mathbb{F}_1\ \textit{GROUPS}$ they are actually associated with. The logical condition *"User is allowed to access the object"* may then be formalised by the following generic Z schema:

$$
\begin{array}{|l}
\hline
\mathit{GrantAccess}[\mathit{OBJECTS}, \mathit{USERS}, \mathit{GROUPS}] \\
\mathit{Object}[\mathit{OBJECTS}, \mathit{USERS}, \mathit{GROUPS}] \\
u? : \mathit{USERS} \\
g? : \mathbb{F}_1\ \mathit{GROUPS} \\
\hline
u? \notin \mathit{uDenied} \\
u? \in u \lor g? \cap g \neq \emptyset \\
\hline
\end{array}
$$

This schema references another schema $\mathit{Object}[\mathit{OBJECTS}, \mathit{USERS}, \mathit{GROUPS}]$ in its declaration part. This *imports* both the variables declared and the predicates specified in $\mathit{Object}[\mathit{OBJECTS}, \mathit{USERS}, \mathit{GROUPS}]$ into $\mathit{GrantAccess}[\mathit{OBJECTS}, \mathit{USERS}, \mathit{GROUPS}]$.

From practical experience we know that it is always helpful to re-translate a mathematical specification and check if this "revised natural-language specification" really meets the requirements. In many cases the new specification will be clearer and more complete than the original text. The Z schema *GrantAccess* states, that we have interpreted the F-C1 requirement cited above as follows:

A user is granted access to an object, if the following conditions are fulfilled:

1. *The user is not contained in the set of users explicitly excluded from object access.*
2. *At least one of the following conditions is fulfilled:*
   (a) *The user is contained in the set of users explicitly allowed to access the object.*
   (b) *When trying to access the object, at least one group possessing access rights to the object is associated with the user.*

Note that we have specified the set $g?$ of groups associated with the user to be non-empty ($\mathbb{F}_1\ M$ is the set of non-empty finite subsets of M). This does not represent a restriction, since always an auxiliary name 'NOGROUP' can be used, if a system allows users without any group association.

GrantAccess specifies only the conditions under which a user is allowed to access the object. However, in the context of IT security we always have to specify *total* operations, where rejecting an unauthorised user is considered a part of the operation. This will be explained in more detail during the formal product evaluation example given below.

Rejecting a user is specified just by negation of the access condition defined in *GrantAccess*:

$$
\begin{array}{|l}
\hline
\mathit{DenyAccess}[\mathit{OBJECTS}, \mathit{USERS}, \mathit{GROUPS}] \\
\mathit{Object}[\mathit{OBJECTS}, \mathit{USERS}, \mathit{GROUPS}] \\
u? : \mathit{USERS} \\
g? : \mathbb{F}_1\ \mathit{GROUPS} \\
\hline
u? \in \mathit{uDenied} \lor u? \notin u \land g? \cap g = \emptyset \\
\hline
\end{array}
$$

5.4.3 Clarification of Ambiguities in Natural-Language Requirements

The F-C1 requirements cited above provide a typical example for ambiguities in natural-language specifications. Close inspection of its relatively “harmless” formalisation shows that we have given an interpretation which deserves discussion. *GrantAccess* implies:

> *There do not exist any groups that explicitly prevent their members from object access, as long as access permission is granted by other groups associated with the user.*

According to our understanding, the text would also admit the following interpretation: A set $gDenied : \mathbb{F}\ GROUPS$ could be introduced, such that object access would not be granted to a user associated with at least one group contained in $gDenied$. The object specification would then look like

$$
\begin{array}{|l}
\hline
XObject[OBJECTS, USERS, GROUPS] \\
o : OBJECTS \\
u : \mathbb{F}\ USERS \\
uDenied : \mathbb{F}\ USERS \\
g : \mathbb{F}\ GROUPS \\
gDenied : \mathbb{F}\ GROUPS \\
\hline
u \cap uDenied = \emptyset \\
g \cap gDenied = \emptyset \\
\hline
\end{array}
$$

The condition to grant object access to a user would be

$$
\begin{array}{|l}
\hline
XGrantAccess[OBJECTS, USERS, GROUPS] \\
XObject[OBJECTS, USERS, GROUPS] \\
u? : USERS \\
g? : \mathbb{F}_1\ GROUPS \\
\hline
u? \notin uDenied \\
g? \cap gDenied = \emptyset \\
u? \in u \lor g? \cap g \neq \emptyset \\
\hline
\end{array}
$$

To motivate that this second interpretation is presumably not the one intended by the ITSEC authors, we can only refer to a concrete implementation “officially” certified to possess F-C1 functionality: The *Berkeley model* for administration of UNIX access rights by means of access control lists allows a process to be associated with more than one group. Therefore modelling $g?$ as a set seems to be adequate. Moreover, the ACL control algorithm (see [42]) is described to calculate the “OR combination” of access rights that the groups contained in $g?$ possess with respect to the object. As a consequence a specific group can never prevent a user from accessing an object, as long as the user is also member of other groups possessing the rights for object access.

It is interesting to note that this textual ambiguity has been overcome in the draft version of the US Federal Criteria [29, CS2-65, b.]. We feel that it is a great advantage that the formal specification at least allows to explain precisely the interpretation we have chosen, so that misunderstandings can be much easier detected than in a natural-language specification.

5.5 A Formal Evaluation Example

In this section we will use the formalised requirement introduced above to illustrate the formal verification process necessary to prove that a product possesses the properties of an ITSEC functionality class. As an example we will consider a UNIX implementation allowing administration of rights by means of *access control lists (ACL)*. This technique is for example described in [110, 42].

5.5.1 UNIX Access Control Lists

The UNIX extensions necessary for this operating system to certify F-C2 functionality[5] include the definition of discretionary access rights by means of ACLs. Every UNIX file is associated with a list of entries of the form

`(user.group, rwx)`

Here, `user` is a subject known to the system, `group` describes a UNIX group associated with `user` and `rwx` stands for the types *read, write, execute* of access right. For `user` and `group` a "don't-care" symbol `%` may be used. If users try to access a file, their identification and the associated groups are compared with the ACL of the file. This comparison algorithm observes the following priorities (1 = highest):

1. `(u.g,rwx)` — specific user, specific group
2. `(u.%, rwx)` — specific user, arbitrary group
3. `(%.g, rwx)` — arbitrary user, specific group
4. `(%.%, rwx)` — arbitrary user, arbitrary group

Any pair `u.g, u.%, %.g, %.%` must occur at most once in the ACL. For example, the entries `(u1.g1, r--), (u1.g1, -w-)` cannot be contained in the ACL at the same time, whereas this is possible for `(u1.g1,r--)` and `(u1.%,---)`.

As soon as a matching ACL entry allowing object access has been found, only matching ACL entries of higher priority are inspected. Access is finally granted if no matching higher-priority entry preventing object access is found. If a user or process is associated with more than one group, the access rights of all matching ACL entries are summarised by means of an OR combination.

Example 5.1 Let a UNIX file have the ACL

`(u2.g2,r--), (u2.%,---), (%.g1,---), (%.g3,r-x), (%.g4,-w-)`

Suppose, user `u1` is associated with group `g1`, user `u2` with groups `g1,g2` and user `u3` with groups `g1,g3,g4`. `u1` cannot access the file, because `(u1,g1)` only matches ACL entry

[5] F-C2 is the next functionality class higher than F-C1.

`(%.g1,---)`, and this entry explicitly forbids object access. `u2` is allowed to read the file because this user is a member of group `g2`; if the `g2` membership were removed from `u2`, access would be denied. User `u3` has full access, since he is a member of groups `g3, g4`; if he would only be in group `g1`, access would be denied, if he would only be in `g4`, he would be allowed to write onto the file.
□

5.5.2 Formalisation of the ACL Model

For being able to prove that the ACL model fulfills the F-C1 requirements, we need a formal specification of the ACL data structures and a mathematical description of the conditions granting file access to a user in the ACL model. This is accomplished by a second Z specification representing a *concrete* security product specification, as implemented by the ACL model.

Let

$$[UXUSERS, UXGROUPS]$$

be the sets of concrete UNIX user identifications and groups, including the don't-care symbols $\%_{users}, \%_{groups}$:

$$\begin{array}{|l} \%_{users} : UXUSERS \\ \%_{groups} : UXGROUPS \end{array}$$

$$[UXFILES]$$

is the concrete representation of the general objects defined in the previous paragraph. Define

$$PERMISSION ::= r \mid w \mid x$$

as the set of UNIX access types.

Using the Z notation, an ACL entry `(u.g,rwx)` can be expressed as a schema

$$\begin{array}{|l} \textit{ACLentry} \\ \hline user : UXUSERS \\ group : UXGROUPS \\ perm : \mathbb{P}\, PERMISSION \\ \hline \end{array}$$

and each complete ACL is a sequence of such entries; i. e. an element of

$$ACLset == \operatorname{seq} ACLentry$$

Each object of our little UNIX universe can now be expressed by a Z schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;UXobject\;\hspace{20em}}}\\
f : UXFILES \\
acl : ACLset \\
\hline
\forall\, i, j : \operatorname{dom} acl \bullet \\
\quad (acl\ i).user = (acl\ j).user \land (acl\ i).group = (acl\ j).group \Rightarrow i = j \\
\hline
\end{array}
$$

Here, $(acl\ i).user$ denotes the `user`-component of the ith ACL entry; dom acl is the domain of all indexes $1, \ldots, \#acl$ numbering the list entries. The predicate part of the schema states that every `user.group` entry may appear only once in the ACL.

The priority ordering of ACL entries introduced above can be formally defined as a relation between pairs `(user,group)`:

$$
\begin{array}{|l}
\_ \leq_{ACL} \_ : (UXUSERS \times UXGROUPS) \leftrightarrow (UXUSERS \times UXGROUPS) \\
\hline
\forall\, u1, u2 : UXUSERS;\ g1, g2 : UXGROUPS \bullet \\
\quad (u1, g1) \leq_{ACL} (u2, g2) \Leftrightarrow \\
\qquad (u1 = u2 \land (g1 = g2 \lor g2 = \%_{groups})) \\
\qquad\quad \lor (u1 \neq u2 \land u2 = \%_{users} \land (g1 = g2 \lor g1 = \%_{groups} \lor g2 = \%_{groups}))
\end{array}
$$

We say that *ACL entry* $(u1, g1, p1)$ *has higher priority than entry* $(u2, g2, p2)$ iff $(u1, g1) \leq_{ACL} (u2, g2)$ holds.

Now for example the formal specification *"UNIX user is granted read access to UNIX file"* can be mathematically described by Z schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;UXgrantReadAccess\;\hspace{15em}}}\\
UXobject \\
u? : UXUSERS \\
g? : \mathbb{F}_1\ UXGROUPS \\
\hline
\exists\, gr : g?;\ a : \operatorname{ran}\ acl \bullet \\
\quad r \in a.perm \land \\
\quad (u?, gr) \leq_{ACL} (a.user, a.group) \land \\
\quad (\forall\, b : \operatorname{ran}\ acl \setminus \{ a \} \bullet \\
\qquad \neg\, ((u?, gr) \leq_{ACL} (b.user, b.group) \land \\
\qquad\quad (b.user, b.group) \leq_{ACL} (a.user, a.group))) \\
\hline
\end{array}
$$

Here, $a : \operatorname{ran}\ acl$ denotes an element contained in the ACL.

In natural-language style, this can be expressed as

> *User u? has read access to the UNIX file, if a group gr* $\in$ *g? is associated with u?, such that*
>
> 1. *There exists a matching ACL entry 'a' granting at least read permission to* $(u?, gr)$.
> 2. *There is no other ACL entry matching with* $(u?, gr)$ *and possessing a higher priority than 'a'.*

Note that the set of groups $g?$ must be non-empty. This reflects the fact that UNIX does not admit users without any associated groups.

Similar schemas *UXgrantWriteAccess* and *UXgrantExecuteAccess* can be defined by replacing the term $r \in a.perm$ by $w \in a.perm$ and $x \in a.perm$, respectively. To specify the condition for full read-, write- and execute-access, we construct the AND-combination

$$
\begin{array}{l}
\mathit{UXgrantFullAccess} \mathrel{\widehat=} \\
\quad \mathit{UXgrantReadAccess} \land \mathit{UXgrantWriteAccess} \land \mathit{UXgrantExecuteAccess}
\end{array}
$$

of all three schemas.

UNIX access control is a deterministic and total operation. As a consequence, the condition for rejecting read access (and analogously write and execute access) is just the negation of the predicate part of *UXgrantReadAccess*, i. e.

$$
\begin{array}{|l}
\hline
\textit{UXdenyReadAccess} \\
\hline
\mathit{UXobject} \\
u? : \mathit{UXUSERS} \\
g? : \mathbb{F}_1\ \mathit{UXGROUPS} \\
\hline
\forall\, gr : g?;\ a : \mathrm{ran}\ acl \bullet \\
\quad r \notin a.perm \lor \\
\quad \neg\,((u?, gr) \le_{ACL} (a.user, a.group)) \lor \\
\quad (\exists\, b : \mathrm{ran}\ acl \setminus \{\, a \,\} \bullet \\
\qquad ((u?, gr) \le_{ACL} (b.user, b.group) \land \\
\qquad\quad (b.user, b.group) \le_{ACL} (a.user, a.group))) \\
\hline
\end{array}
$$

Analogously, *UXdenyWriteAccess, UXdenyExecuteAccess* are defined, so giving only limited access or none at all is specified by

$$
\begin{array}{l}
\mathit{UXdenyFullAccess} \mathrel{\widehat=} \\
\quad \mathit{UXdenyReadAccess} \lor \mathit{UXdenyWriteAccess} \lor \mathit{UXdenyExecuteAccess}
\end{array}
$$

5.5.3 Instantiation of the Generic Specification

To be able to verify the concrete specifications *UXgrantFullAccess* and *UXdenyFullAccess* of the product against the abstract specifications, we first have to instantiate the generic data types used in *GrantAccess* and *DenyAccess* with the concrete data structures of the product. To this end, we create a new F-C1 object specification by instantiating it with the concrete UNIX data types *UXFILES, UXUSERS, UXGROUPS*:

$$\mathit{FC1objectForUnix} \mathrel{\widehat=} \mathit{Object}[\mathit{UXFILES}, \mathit{UXUSERS}, \mathit{UXGROUPS}]$$

5.5.4 Abstraction Relation Between F-C1 Model and ACL Model

The process of replacing an abstract specification (in our case the formal F-C1 specification, instantiated with the concrete UNIX data types) by a concrete one (the ACL model) is a refinement step. The most important task for the refinement process is to relate the abstract and concrete data structures by means of an *abstraction relation.*

$$
\begin{array}{|l}
\textit{Abstraction} \\
\hline
\textit{FC1objectForUnix} \\
\textit{UXobject} \\
\hline
o = f \\[1ex]
\forall\, a : \operatorname{ran} acl \bullet \\
\quad (a.perm = \{\, r, w, x \,\} \lor a.perm = \emptyset) \land \\
\quad (a.user, a.group) \in \\
\qquad ((\mathit{UXUSERS} \setminus \{\%_{users}\}) \times \{\%_{groups}\}) \cup \\
\qquad\quad (\{\%_{users}\} \times (\mathit{UXGROUPS} \setminus \{\%_{groups}\})) \\[1ex]
u = \{a : \operatorname{ran} acl \mid a.user \neq \%_{users} \land a.perm = \{\, r, w, x \,\} \bullet a.user\} \\
\mathit{uDenied} = \{a : \operatorname{ran} acl \mid a.user \neq \%_{users} \land a.perm = \emptyset \bullet a.user\} \\
g = \{a : \operatorname{ran} acl \mid a.group \neq \%_{groups} \land a.perm = \{\, r, w, x \,\} \bullet a.group\} \\
\hline
\end{array}
$$

The five predicates defined in the abstraction relation have the following meaning[6]:

1. *The "logical contents" of the objects – i. e. the interpretation of the UNIX file contents – is identical on both levels of abstraction.*
2. *While the abstract F-C1 requirement only knows one access type (i. e. full object access), the concrete state space UXobject allows to express different types of access rights (read, write, execute). As a consequence, only a subset of possible ACLs is related to the F-C1 model, namely those where each entry either completely grants or completely denies object access. Moreover, to implement the abstract F-C1 state space, only those ACLs are needed where each entry has a don't-care symbol in either the* `user`*- or the* `group`*-component.*
3. *The abstract set of users explicitly allowed to access the object is implemented by the ACL entries of the form* `(u.%,rwx)` *associated with the UNIX file.*
4. *The abstract set of users explicitly excluded from object access is implemented by the ACL entries of the form* `(u.%,---)`*.*
5. *The abstract set of groups granting object access is implemented by the ACL entries of the form* `(%.g,wrx)`*.*

To prove that we have chosen a "reasonable" abstraction relation it must be shown that the complete abstract state space can be modelled by means of the concrete one. This is a consequence of the next theorem.

Theorem 14 (Suitability of the Abstraction Relation) *Schema* Abstraction *specifies a well-defined partial surjective retrieve function*

$$\mid\ \phi : \mathit{UXobject} \twoheadrightarrow \mathit{FC1objectForUnix}$$

[6] Note that the Z notation for sets has been used: The elements contained in a set are defined by the expressions after the •-marker.

Proof.
Clearly the predicate part of schema *Abstraction* specifies a well-defined partial function ϕ. It remains to show that ran $\phi = FC1objectForUnix$.

(a) ran $\phi \subseteq FC1objectForUnix$*:* Since by definition ran ϕ is a subset of the type of $FC1objectForUnix$, we only have to prove that

$$\forall\, y : \mathrm{ran}\ \phi \bullet y.u \cap y.uDenied = \emptyset$$

since this is the condition of the predicate part of $FC1objectForUnix$. Now let $y = \phi(x)$ and suppose $v \in y.u$ and $w \in y.uDenied$. According to the definition of ϕ and because $x \in \mathrm{dom}\ \phi$, there exist ACL entries $a, b \in \mathrm{ran}\ x.acl$, such that

$$v = a.user \wedge a.perm = \{r, w, x\} \wedge a.group = \%_{groups}$$

and

$$w = b.user \wedge b.perm = \emptyset \wedge b.group = \%_{groups}$$

Obviously, a and b are distinct entries in $x.acl$. Since they do not differ in the *group*-component, the predicate part of *UXobject* implies $a.user \neq b.user$. Therefore, $v \neq w$ and we have shown that $y.u$ and $y.uDenied$ have an empty intersection. As a consequence, y is an element of $FC1objectForUnix$.

(b) $FC1objectForUnix \subseteq$ ran ϕ*:* Let $y \in FC1objectForUnix$. Since $y.u, y.uDenied, y.g$ are finite sets, there exist bijective functions

$$\begin{aligned} &h : 1 \mathinner{..} \#(y.u) \rightarrowtail\!\!\!\!\twoheadrightarrow y.u \\ &k : 1 \mathinner{..} \#(y.uDenied) \rightarrowtail\!\!\!\!\twoheadrightarrow y.uDenied \\ &\ell : 1 \mathinner{..} \#(y.g) \rightarrowtail\!\!\!\!\twoheadrightarrow y.g \end{aligned}$$

Now define

$$\begin{aligned} &x.f = y.o \\ &x.acl = s \frown t \frown v \end{aligned}$$

such that the sequences s, t, v satisfy $\#s = \#(y.u), \#t = \#(y.uDenied), \#v = \#(y.g)$ and[7]

$$\begin{aligned} &(s\ i).user = h(i) \wedge (s\ i).group = \%_{groups} \wedge (s\ i).perm = \{r, w, x\} \\ &(t\ i).user = k(i) \wedge (t\ i).group = \%_{groups} \wedge (t\ i).perm = \emptyset \\ &(v\ i).user = \%_{users} \wedge (v\ i).group = \ell(i) \wedge (v\ i).perm = \{r, w, x\} \end{aligned}$$

Then x is in dom ϕ and $\phi(x) = y$ by construction.
□

[7] $(s\ i).user$ denotes the *user*-component of the ith entry in s.

5.5.5 Verification of the Refinement

To finish our evaluation example, we present a refinement proof for the UNIX access control operation. In the context of *sequential systems*, the following conditions have to be verified in order to prove the correctness of a refinement step (see [117, p. 200]).

Initialisation Condition: The initialisation of the concrete system always results in an initial state related to an initial state of the abstract system by means of the abstraction relation.

Safety Condition: Whenever the pre-state of the abstract system allows an abstract operation, any concrete state related to the abstract state by means of the abstraction relation will admit the corresponding concrete operation.

Lifeness Condition: If abstract and concrete pre-states are related via the abstraction relation, then execution of both an abstract and the corresponding concrete operation will also lead to related after-states.

Since we did not define anything about system start, we will skip the initialisation condition here. In contrast to other applications of function refinement, we have to analyse both the *GrantAccess* and the *DenyAccess* operations *in combination* for the safety condition: If the refinement rules were only applied to the *GrantAccess* part, then weakening the precondition for object access would be admissible in the sense of general refinement, but certainly not meet the desired security requirements. Moreover, we want the *grant/deny*-decision to be deterministic. Therefore the following extended version of the safety condition for general function refinement seems appropriate:

Theorem 15 (Safety Conditions for Secure Functions)

1. *The concrete GrantAccess and DenyAccess conditions partition the the domain of retrieve function ϕ into two disjoint sets. As a consequence, the decision* "Access granted/denied" *is always deterministic.*

$$[u? : \mathit{UXUSERS};\ g? : \mathbb{F}_1\ \mathit{UXGROUPS}] \land \mathit{Abstraction} \vdash \\ \neg\ (\mathit{UXgrantFullAccess} \land \mathit{UXdenyFullAccess})$$

2. *Whenever object access is granted in the abstract system, the same holds for the concrete system:*

$$\mathit{GrantAccess}[\mathit{UXFILES}, \mathit{UXUSERS}, \mathit{UXGROUPS}] \land \mathit{Abstraction} \vdash \\ \mathit{UXgrantFullAccess}$$

3. *Whenever object access is denied in the abstract system, the same holds for the concrete system:*

$$\mathit{DenyAccess}[\mathit{UXFILES}, \mathit{UXUSERS}, \mathit{UXGROUPS}] \land \mathit{Abstraction} \vdash \\ \mathit{UXdenyFullAccess}$$

Proof.
Statement 1. is obvious, since the predicate part of *UXdenyFullAccess* is just the negation of *UXgrantFullAccess*. We present the proof for condition 3.; the proof for 2. is similar.

We start with the user-object configuration in the abstract F-C1 model. The abstract object is instantiated by schema *FC1objectForUNIX*. Let $u?$ be the user's identification, $g?$ the set of groups associated with $u?$. According to the assumption, $(u?, g?)$ satisfy the predicate part of schema

$$DenyAccess[UXOBJECTS, UXUSERS, UXGROUPS]$$

so

$$u? \in uDenied \lor u? \notin u \land g? \cap g = \emptyset$$

Case 1. $u? \in uDenied$

Using schema *Abstraction*, we can relate the abstract set *uDenied* with a concrete set of ACL entries; it follows that

$$u? \in \{a : \operatorname{ran} acl \mid a.user \neq \%_{users} \land a.perm = \emptyset \bullet a.user\}$$

Now let $a \in \operatorname{ran} acl$ such that $a.user = u?$ and $a.perm = \emptyset$. Since the access control list acl containing a is related to the abstract model via schema *Abstraction*, it follows from the predicate part of this schema that $a.group = \%_{groups}$. As a consequence the definition of $\leq_{ACL}$ implies

$$\forall gr : g? \bullet (u?, gr) \leq_{ACL} (a.user, a.group)$$

Now chose any entry $b \in \operatorname{ran} acl$ such that

$$(u?, gr) \leq_{ACL} (b.user, b.group) \leq_{ACL} (a.user, a.group)$$

Definition of $\leq_{ACL}$-priorities implies that $b.user = a.user = u?$. Schema *Abstraction* then implies that $b.group = a.group = \%_{groups}$. Therefore, according to schema *UXobject*, we have $b = a$.

Summarising these facts, we have shown that

$$\begin{array}{l}
\forall gr : g? \bullet \exists a : \operatorname{ran} acl \bullet \\
\quad a.perm = \emptyset \land \\
\quad (u?, gr) \leq_{ACL} (a.user, a.group) \land \\
\quad (\forall b : ran\ acl \setminus \{ a \} \bullet \\
\qquad \neg ((u?, gr) \leq_{ACL} (b.user, b.group) \land \\
\qquad\qquad (b.user, b.group) \leq_{ACL} (a.user, a.group)))
\end{array}$$

holds. This means that the pair $(u?, g?)$ fulfills the condition of schema *UXdenyFullAccess*, because the predicate parts of all three schemas *UXdenyReadAccess, UXdenyWriteAccess, UXdenyExecuteAccess* are fulfilled.

Case 2. $u \notin uDenied \wedge u? \notin u \wedge g? \cap g = \emptyset$

Take any $gr \in g?$ and $a \in \text{ran } acl$ such that $a.user \neq \%_{users} \wedge a.perm = \{r, w, x\}$. $u? \notin u$ implies

$$\neg\, ((u?, gr) \leq_{ACL} (a.user, a.group))$$

Now take any $b \in \text{ran } acl$ such that $b.group \neq \%_{groups} \wedge b.perm = \{r, w, x\}$. Since $g? \cap g = \emptyset$, we have

$$\neg\, ((u?, gr) \leq_{ACL} (b.user, b.group))$$

As a consequence, the ACL is empty or all possible entries with $\{r, w, x\}$-permission are incomparable with $(u?, gr)$, so again the predicate parts of *UXdenyReadAccess, UXdenyWriteAccess, UXdenyExecuteAccess* are fulfilled.

Cases 1 and 2 imply that also in the concrete ACL model permission to access the object is denied to this user.
□

We skip the liveness condition, because we did not specify anything about the object operations themselves (READ, WRITE,...), but only presented the preconditions under which the *grant*-part or *deny*-part of each operation should operate.

5.6 Discussion and Future Work

In this chapter we have sketched a method applicable for the formal specification of generic ITSEC functionality classes. The technique described would be equally well suited to give mathematical specifications of the functional requirements in other related standards, e. g. the US Federal Criteria [28, 29]. I am convinced that such formalisations could serve as a very helpful addendum to the natural-language document in order to clarify ambiguous requirements and provide a well-defined and consistent reference for evaluation and certification of IT security products. The generic nature of the specifications described would make them also suitable to serve as re-usable input documents for the formal specifications of new high-quality products to be developed, thus reducing the amount of work to be spent on the specification phase. Several aspects of the approach described above deserve discussion and require extensions:

Alternatives to the Model-Oriented Approach Using generic Z specifications for the formalisation of ITSEC requirements, we have chosen a *model-oriented* approach where data structures – though generic in several aspects – become visible from the beginning.

Example 5.2 Schema $Object[OBJECTS, USERS, GROUPS]$ uses generic data types, but explicitly states that the objects to be protected by the security mechanisms should be regarded as 4-tuples $(o, u, uDenied, g)$, where $u, uDenied$ and g have powerset types. If a concrete system choses to use different data structures – like the ACLs for UNIX –, the connection has to be made explicit by means of the abstraction relation.
□

If we used *algebraic* specification techniques on this level, this might allow to cover a wider class of system types by the generic specification.

Example 5.3 *Object* might be characterised by a set of functions like

$$Access : Object \times USERS \times \mathbb{F}_1\, GROUPS \rightarrow \{grant, deny\}$$

$$AlwaysDenied : Object \times USERS \rightarrow BOOL$$

using axioms like

$$Access(o?, u?, g?) = grant \Rightarrow AlwaysDenied(o?, u?) = false$$

This leaves the internal structure of *Object* completely unrestricted, but still captures the essential object information to be used by the security mechanisms.
□

Behavioural and Architectural Aspects The approach illustrated above solves only one out of three problems to be covered by trustworthy IT security products, namely the functionality of security mechanisms. The two other aspects are

- **Secure Dynamic Behaviour:** A security mechanism should grant the degree of protection required, regardless of other activities executed in the system in parallel. This problem of *non-interference* depends on the dynamic behaviour of the system. A most critical issue is the difficulty that non-interference will not always be preserved under the application of standard refinement techniques. This has been investigated quite extensively (see, for example, [48, 32, 33]), but from my point of view the most promising approach has been described only recently by Roscoe, Woodcock and Wulf [101]: The isolated access operations can be described by means of Z as shown above, but their parallel execution is specified by means of an embedding of the Z specifications into CSP. Proofs about interference freedom can be designed as refinement proofs in the CSP process algebra. The authors relate the absence of undesired information flow to the absence of nondeterminism. They show that non-interference will be preserved under refinement, if a certain degree of determinism can be guaranteed in the system.
- **Secure Architecture:** It is well known that the general refinement approach does not always lead to secure implementations, because again refinement might introduce security leaks, this time in the form of *covert channels*, [50]. These are channels which have been introduced during the refinement process and are therefore not covered by the security mechanisms designed on specification level. Intruders may use these "lower-level" channels to corrupt data or obtain secret information. This problem is comparable to the one to be encountered in the development of fault-tolerant systems, where certain choices of refinement may force the designer to alter the original fault-hypotheses. To solve these difficulties it may be useful to develop *standard architectures* for IT security products and define the requirements for security mechanisms based on this architecture. An approach for the development of such architectural standard models has been described in [6].

Conformance Tests Defined by Generic Specifications The design of trustworthy test suites considerably increases the evaluation costs for IT security products. Therefore it will be necessary to define *conformance tests* which are designed in a way to be mapped easily to specific products. Such test suites are well known, for example, in the fields of telecommunications and compiler validation, but do not exist for IT security. A promising approach could be the design of "generic tests cases" to be instantiated for the concrete product along with the generic specification.

Bibliography

[1] DA Airbus Industrie No 2370M1F000101: *Cabin Intercommunication Data System System Specification, A330/340, CIDS* Volume I, Issue 5 (1993).

[2] K. R. Apt, L. Bougé and Ph. Clermont: Two Normal Form Theorems for CSP Programs. *Information Processing Letters* (1987) 26:165-171.

[3] K. R. Apt, N. Francez and W. P. de Roever: A Proof System for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems* (1980) 9: 359-385.

[4] K. R. Apt: Correctness Proofs of Distributed Termination Algorithms. *ACM Trans. on Progr. Lang. and Syst.* (1986) 8: 388-405.

[5] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, Berlin Heidelberg New York (1991).

[6] R. Atzmüller: *Das REMO Referenzmodell.* BSI (1994).

[7] F. Belli, K. Echtle and W. Görke: Methoden und Modelle der Fehlertoleranz. *Informatik Spektrum* (1986) 9: 68-81.

[8] A. Benveniste and G. Berry. *The Synchronous Approach to Reactive and Real-Time Systems*, in IEEE-Proceedings *"Another Look at Real-Time Programming"*, 1992.

[9] P. Bieber and N. Boulahia-Cuppens: Formal Development of Authentication Protocols. In D. Till (Ed.): *6th Refinement Workshop.* Proceedings of the 6th Refinement Workshop, organised by BCS FACS, London, 5-7 January 1994, Springer-Verlag, Berlin Heidelberg New York (1994) 80-102.

[10] D. Björner, H. Langmaack and C. A. R. Hoare: *ProCoS – Provably Correct Systems.* Esprit BRA 3104. Technical University of Denmark, Lyngby (1993).

[11] E. Brinksma: A theory for the deriviation of tests. In P. H. J. van Eijk, C. A. Vissers and M. Diaz (Eds.): *The Formal Description Technique LOTOS.* Elsevire Science Publishers B. V. (North-Holland), (1989) 235-247.

[12] M. Broy: Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing* (1991) 3: 21-57

[13] F. Christian: Correct and Robust Programs. *IEEE Transactions on Software Engineering* (1984) 10: 163-174.

[14] F. Christian: A Rigorous Approach to Fault-Tolerant Programming. *IEEE Transactions on Software Engineering* (1985) 11: 23-31.

[15] F. Christian: Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM* (1991) 34: 57-78.

[16] E. M. Clarke, O. Grümberg and D. E. Long: Model Checking and Abstraction. In *Proc. of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL '92)* (1992).

[17] *Common Criteria for Information Technology Security Evaluation.* Draft Version 0.6. CCEB-94/034.

[18] J. Davies: *Specification and Proof in Real-Time CSP.* Cambridge University Press (1993).

[19] D. W. Davies: Protection. In B. E. Lampson, M. Paul, H. J. Siegert (Eds.): *Distributed Systems.* Springer-Verlag, Berlin Heidelberg New York (1983) 211-245.

[20] T. DeMarco: *Structured Analysis and System Specification.* Prentice-Hall International, Englewood Cliffs NJ (1979).

[21] Using Z and DST-fuzz, an Introduction. DST Deutsche System-Technik GmbH, Kiel (1993).

[22] RTCA DO178B: *Development considerations in airborne computer systems.* (1993).

[23] E. W. Dijkstra: Self-stabilizing systems in spite of distributed control. *Communications of the ACM* (1974) 17: 643-644.

[24] *DIN ISO 9000-3: Qualitätsmanagement- und Qualitätssicherungsnormen – Leitfaden für die Anwendung von ISO 9001 auf die Entwicklung, Lieferung und Wartung von Software.* DIN Deutsches Institut für Normung e. V., Beuth Verlag Berlin (1990).

[25] ELPRO LET GmbH: *Programmablaufplan - Bahnübergang.* ELPRO LET GmbH (1994).

[26] ELPRO LET GmbH: *Radio-based Integrated Control Information and Signalling System for Railways.* Proposal for EU Telematics Applications Programme 'Telematics for Transport Sector', ELPRO LET GmbH (1995).

[27] Formal Systems: *Failures Divergence Refinement.* User Manual and Tutorial Version 1.4. Formal Systems (Europe) Ltd (1994).

[28] Federal Criteria for Information Technology Security, Volume I, Version 1.0. U. S. National Institute of Standards and Technology & National Security Agency (1992).

[29] Federal Criteria for Information Technology Security, Volume II, Version 1.0. U. S. National Institute of Standards and Technology & National Security Agency (1992).

[30] M.-C. Gaudel: Testing can be formal, too. In P. D. Mosses, M. Nielsen and M. I. Schwartzbach (Eds.): *Proceedings of TAPSOFT '95: Theory and Practice of Software Development.* Aarhus, Denmark, May 1995, Springer-Verlag, Berlin Heidelberg New York (1995).

[31] W. W. Gibbs: Software chronisch mangelhaft. *Spektrum der Wissenschaft* (1994) 12: 56-63.

[32] J. Graham-Cunning and J. W. Sanders: On the Refinement of Non-interference. In *Proceedings of the Computer Security Foundations Workshop IV* IEEE (1991).

[33] J. Graham-Cunning: Some Laws of Non-interference. In *Proceedings of the Computer Security Foundations Workshop V* IEEE (1992).

[34] M. Grochtmann and K. Grimm: Classification Trees for Partition Testing. *Software Testing, Verification and Reliability* (1993) 3: 63-82.

[35] D. Harel, A. Pnueli, J. Pruzan-Schmidt and R. Sherman. *On the formal semantics of Statecharts.* In *Proceedings Symposium on Logic in Computer Science*, (1987) 54-64.

[36] D. Harel. *On visual formalisms.* Communications of the ACM (1988) 31: 514-530.

[37] D. J. Hatley and I. A. Pirbhai: *Strategies for Real-Time System Specification.* Dorset House, New York (1987).

[38] M. Heisel and D. Weber-Wulff: Korrekte Software: Nur eine Illusion? *Informatik Forschung und Entwicklung* (1994) 9: 192-200

[39] M. C. Hennessy: *Algebraic Theory of Processes.* MIT Press (1988).

[40] M. C. Hennessy and A. Ingólfsdóttir: Communicating Processes with Value-passing and Assignments. *Formal Aspects of Computing* (1993) 5: 432-466

[41] H. M. Hörcher: Improving Software Tests using Z Specifications. To appear in J. P. Bowen and M. G. Hinchey (Eds.): *ZUM '95: 9th International Conference of Z Users*, LNCS, Springer-Verlag, Berlin Heidelberg New York (1995).

[42] HP-UX System Security. Hewlett-Packard Company (1991).

[43] C. Huizing, R. Gerth and W.-P. de Roever. *Modelling statecharts behaviour in a fully abstract way.* In *Proc. 13th CAAP*, LNCS 299, Springer-Verlag, Berlin Heidelberg New York (1988) 271-294.

[44] C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall International, Englewood Cliffs NJ (1985).

[45] C. Huizing and R.T. Gerth. *Semantics of Reactive Systems in Abstract Time*, in *"Real-Time: Theory in Practice"*, proceedings of a REX workshop, June 1991, Mook, edited by J.W. de Bakker, W.-P. de Roever, G. Rozenberg, LNCS 600, Springer-Verlag, Berlin Heidelberg New York, 1992.

[46] Inmos Ltd., *Occam 2 Reference Manual.* Prentice-Hall International, Englewood Cliffs NJ, (1988).

[47] Information Technology Security Evaluation Criteria (ITSEC). Version 1.2. EGKS-EWG-EAG, Brussels • Luxemburg (1991).

[48] J. Jacob: Security Specifications. In *IEEE Symposium on Security and Privacy '88* IEEE (1988).

[49] P. Jalote: Fault tolerant processes. *Distributed Computing* (1989) 3: 187-195.

[50] M. K. Joseph: Integration Problems in Fault Tolerant, Secure Computer Design. In A. Avižienis, J. C. Laprie (Eds.): *Dependable Computing for Critical Applications.* Springer-Verlag, Berlin Heidelberg New York (1991) 347-364.

[51] He Jifeng and C. A. R. Hoare: Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing* (1987) 2: 1-12.

[52] C. B. Jones: *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs NJ (1986).

[53] E. Jonsson and T. Olovsson: Security in a Dependability Perspective. In *Proc. of Nordic Seminar on Dependable Computing NSDCS '94.*

[54] Y. Kesten and A. Pnueli: Timed and hybrid statecharts and their textual representation. In J. Vytopil (Ed.) *Formal Techniques in Real-Time and Fault-Tolerant Systems.* Springer-Verlag, Berlin Heidelberg New York (1992) 591-619.

[55] B. W. Lampson, M. Paul and H. J. Siegert (Eds.): *Distributed Systems.* Springer-Verlag, Berlin Heidelberg New York (1983).

[56] J. C. Laprie: Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proc. 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15)*, (1985).

[57] J. C. Laprie et al.: *Dependability: Basic Concepts and Terminology.* Springer-Verlag, Berlin Heidelberg New York (1992).

[58] J. C. Laprie: Dependability: a unifying concept for reliable, safe and secure computing. In *Proc. of the 12th IFIP World Computer Congress, Madrid, Spain.* (1992).

[59] P. A. Lee and T. Anderson: *Fault tolerance: Principles and practice.* Springer-Verlag, Berlin Heidelberg New York (1990).

[60] J. Loeckx and K. Sieber: *The Foundations of Program Verification.* 2nd Ed.-Wiley,Teubner, Stuttgart (1987).

[61] T. F. Lunt and R. A. Whitehurst: *The Seaview Formal Top Level Specification.* A007: Final Report, Volume 3A. SRI International, Menlo Park, CA, (1989).

[62] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, Berlin Heidelberg New York (1992).

[63] E. Mikk: Compilation of Z Specifications into C for Automatic Test Result Evaluation. To appear in J. P. Bowen and M. G. Hinchey (Eds.): *ZUM '95: 9th International Conference of Z Users*, LNCS, Springer-Verlag, Berlin Heidelberg New York (1995).

[64] R. Milner: *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs NJ (1989).

[65] Requirements for the procurement of safety critical software in defence equipment, *Interim Defence Standard 00-55*, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown St., Glasgow G2 8EX, May (1989).

[66] C. C. Morgan: *Programming from Specifications.* Prentice-Hall International, Englewood Cliffs NJ (1990).

[67] C. C. Morgan, K. A. Robinson and P. H. B. Gardiner: On the refinement calculus. *Oxford University Computing Laboratory.* PRG-70 (1989).

[68] M. Müllerburg: Systematic Testing: a Means for Validating Reactive Systems. In *EuroSTAR'94: Proceedings of the 2nd European Intern. Conf. on Software Testing, Analysis&Review.* British Computer Society, (1994).

[69] R. de Nicola and M. C. Hennessy: Testing Equivalence for Processes. *Journal of Theoretical Computer Science* (1983)34: 83-133

[70] J. Nordahl: *Specification and Design of Dependable Communicating Systems.* Thesis, Department of Computer Science, Technical University of Denmark, Lyngby (1992).

[71] E.-R. Olderog: *Nets, Terms and Formulas.* Cambridge University Press (1991).

[72] J. Peleska: A characterization for isometries and conformal mappings of pseudo-Riemannian manifolds. *Aequationes Mathematicae*(1984)27: 20-31.

[73] J. Peleska: *Das PST-NET Dopelrechnersystem.* Technical Report, Philips GmbH, Unternehmensbereich Systeme und Sondertechnik (1989).

[74] J. Peleska: Design and verification of fault tolerant systems with CSP. *Distributed Computing* (1991) 5: 95-106.

[75] J. Peleska: Formale Methoden beim Entwurf ausfallsicherer, verteilter Systeme. In Lippold, Schmitz (Eds.): *Sicherheit in netzgestüzten Informationssystemen.* Proceedings des BIFOA-Kongresses SECUNET '92, Vieweg (1992) 293-308.

[76] J. Peleska: CSP, Formal Software-Engineering and the Development of Fault-Tolerant Systems. In Vytopil (Ed.): *Formal Techniques in Real-Time and Fault-Tolerant Systems.* Kluwer Academic Publishers (1993) 167-207.

[77] J. Peleska: Formale Spezifikation generischer ITSEC Funktionalitätsklassen. In Reichel (Ed.): Informatik, Wirtschaft, Gesellschaft. 23. GI Jahrestagung, Springer-Verlag, Berlin Heidelberg New York (1993) 354-364.

[78] J. Peleska: On a Unified Approach for the Development of Fault-Tolerant and Secure Systems. In *Proceedings of the Nordic Seminar on Dependable Computing '94* (invited keynote).

[79] U. Hamer, H. M. Hörcher and J. Peleska: Safer Software - an Introduction into Formal Software Engineering. Technical Report, DST (1993).

[80] J. Peleska: Formales Software-Engineering mit strukturierten Methoden und Z: nachweisbare Korrektheit für sicherheits-relevante Anwendungen. In Scheibel (Ed.): *Software-Entwicklung - Methoden, Werkzeuge, Erfahrungen '93.* Technische Akademie Esslingen (1993) 753-762.

[81] J. Peleska, C. Huizing and C. Petersohn: A Comparison of Ward&Mellor's Transformation Schema with State-&Activitycharts. Computing Science Note 94/11, Eindhoven University of Technology (1994) – Submitted for publication in *IEEE Transactions on Software Engineering.*

[82] C. Petersohn, C. Huizing, J. Peleska and W. P. de Roever: Formal Semantics for Ward&Mellor's Transformation Schemas. In D. Till (Ed.): *6th Refinement Workshop.* Proceedings of the 6th Refinement Workshop, organised by BCS FACS, London, 5-7 January 1994, Springer-Verlag, Berlin Heidelberg New York (1994) 14-41.

[83] H. M. Hörcher and J. Peleska: The Role of Formal Specifications in Software Test. Tutorial, held at the FME '94.

[84] J. Peleska: Simulation und Wirklichkeit - Stellungnahme aus Sicht der Industrie. In *Computersimulation: (k)ein Spiegel der Wirklichkeit.* Bundesamt für Sicherheit in der Informationstechnik, SecuMedia-Verlag (1994) 163-170.

[85] U. Hamer and J. Peleska: Z Applied to the A330/340 CIDS Cabin Communication System. In M. Hinchey, J. Bowen (Eds.) *Applications of Formal Methods.* Prentice-Hall International, Englewood Cliffs NJ (1995).

[86] J. Peleska: *Bahnübergangssteuerung Straßenbahn – ELPRO LET GmbH: Prüfspezifikation für formale Verifikation und automatisierte Testdurchführung.* JP Software-Consulting (1994).

[87] J. Peleska: *Bahnübergangssteuerung Straßenbahn – ELPRO LET GmbH: Sicherheitsspezifikation und BUE-Spezifikation.* JP Software-Consulting (1994).

[88] A. Baer, B. Krieg-Brückner, E.-R. Olderog and J. Peleska: *Universelle Entwicklungsumgebung für Formale Methoden (UniForM).* Vorhaben im Fördergebiet 'Softwaretechnologie' des BMBF (1995).

[89] J. Peleska: *Trustworthy Tests for Reactive Systems – Automation of Real-Time Testing.* In preparation, JP Software-Consulting (1997).

[90] J. Peleska: *Testautomatisierung für diskrete Steuerungen Anwendung: Bahnübergangssteuerung Abschlußbericht Phase 1* JP Software-Consulting (1995).

[91] J. Peleska and M. Siegel From Testing Theory to Test Driver Implementation. In M.-C. Gaudel and J. Woodcock (Eds.): *FME '96: Industrial Benefit and Advances in Formal Methods.* LNCS 1051, Springer-Verlag, Berlin Heidelberg New York (1996) 538-556.

[92] J. Peleska: Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In M.-C. Gaudel and J. Woodcock (Eds.): *FME '96: Industrial Benefit and Advances in Formal Methods.* LNCS 1051, Springer-Verlag, Berlin Heidelberg New York (1996) 39-59.

[93] J. Peleska and M. Siegel: Test Automation of Safety-Critical Reactive Systems. To appear in *South African Computer Jounal*, (1997).

[94] G. Plotkin: *An operational semantics for CSP.* In *Proceedings of the IFIP Conference on the Formal Description of Programming Concepts II,* North Holland (1983) 199-225.

[95] N. Pohlmann: Sicherheit der Kommunikation in Netzen. In: *Proceedings of the 3. Deutscher IT-Sicherheitskongreß des BSI 1993.* Bundesamt für Sicherheit in der Informationstechnik. SecuMedia, Ingelheim (1994) 154-180.

[96] A. P. Ravn and V. Stavridou: *Criteria for specification and programing languages for engineering safety-critical software.* Technical report, Technical University of Denmark (ID/DTH) (1995).

[97] A. P. Ravn and V. Stavridou: *Specification and development of safety-critical software: An assessment of MOD draft Standard 00-55.* Technical report, Technical University of Denmark (ID/DTH) (1995).

[98] G. Reed and A. W. Roscoe: A timed model for Communicating Sequential Processes. In: *Proc. 13th International Colloquium on Automata, Languages and Programming.* LNCS 226, Springer-Verlag, Berlin Heidelberg New York (1986) 314-323.

[99] A. W. Roscoe and G. Barret: Unbounded Nondeterminism in CSP. In *MFPS '89*, LNCS 298, Springer-Verlag, Berlin Heidelberg New York (1989).

[100] A. W. Roscoe: Model-Checking CSP. In *A Classical Mind, Essays in Honour of CAR Hoare.* Prentice-Hall International, Englewood Cliffs NJ (1994).

[101] A. W. Roscoe, J. C. P. Woodcock and L. Wulf: Non-interference through Determinism. Oxford University Computing Laboratory (1994).

[102] J. Rushby: Critical system properties: survey and taxonomy. *Reliability Engineering and System Safety* 43: 189-219 (1994).

[103] H. Schepers: Terminology and Paradigms for Fault Tolerance. In Vytopil (Ed.): *Formal Techniques in Real-Time and Fault-Tolerant Systems.* Kluwer Academic Publishers (1993) 3-31.

[104] H. Schepers: *Fault Tolerance and Timing of Distributed Systems.* Thesis, Eindhoven University of Technology (1994).

[105] R. D. Schlichting and Fred B. Schneider: Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems* (1983) 1: 222-238.

[106] M. Schneider: Self-stabilization. *ACM Computing Surveys* (1993) 25: 45-67.

[107] K. G. Shin and P. Ramanathan: Real-Time Computing: A new Discipline of Computer Science and Engineering. *Proceedings of the IEEE* (1994) 82: 6-23.

[108] M. J. Spivey. *The Z Notation.* Prentice-Hall International, Englewood Cliffs NJ (1992).

[109] M. E. Stickel: Elimination of Inference Channels by Optimal Upgrading. In *IEEE Symposium on Security and Privacy '94* IEEE (1994).

[110] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall International, Englewood Cliffs NJ (1992).

[111] U. S. Department of Defense Trusted Computer System Evaluation Criteria (TCSEC). DoD 5200.28-STD (1985).

[112] Der Bundesminister des Innern: *Planung und Durchführung von IT-Vorhaben – Vorgehensmodell.* KBSt Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (1992). English version available from IABG Industrieanlagen Betriebsgesellschaft, Ottobrunn (1992).

[113] P. T. Ward: *The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. IEEE Transactions on Software Engineering* (1986) SE-12: 198-210.

[114] P. T. Ward and S. J. Mellor: *Structured Development for Real-Time Systems.* (3 vols), Yourdon Press Computing Series, Prentice-Hall International, Englewood Cliffs NJ Cliffs, 1985.

[115] J. Wegener, R. Pitschinetz, K. Grimm and M. Grochtmann: TESSY - Yet Another Computer-Aided Software Testing Tool? Daimler-Benz AG, Forschung und Technik, Berlin (1994).

[116] R. A. Whitehurst and T. F. Lunt: The Seaview Verification Effort. In *Proceedings of the 12th National Computer Security Conference.* Baltimore, MD, (1989).

[117] J. B. Wordsworth : *Software Development with Z.* Addison-Wesley,Wokingham (U.K.) (1992).

[118] Zhiming Liu, E. V. Sørensen, A. P. Ravn and Chaochen Zhou: Towards a Calculus of System Dependability. *Journal of high integrity systems* (1994) 1: 49-65 .

[119] D. Zöbel: Normalform-Transformationen für CSP-Programme. *Informatik Forsch. Entw.* (1988) 3: 64-76.

[120] J. Zwiers: *Compositionality, Concurrency and Partial Correctness.* LNCS 321, Springer-Verlag, Berlin Heidelberg New York (1989).

Appendix A. Glossary of Symbols

Functions

Symbol	Meaning
$f : X \rightarrow Y$	function f with domain X and range Y
$f : X \nrightarrow Y$	partial function f
$f : X \twoheadrightarrow Y$	surjective function f
$f : X \rightarrowtail Y$	injective function f
$f : X \rightarrowtail\kern-1.1em\twoheadrightarrow Y$	bijective function f
$\operatorname{dom} f$	domain of f
$\operatorname{ran} f$	range of f
$f \oplus g$	function f overridden on dom g by values of g

Traces

Symbol	Meaning	Example
$s = \langle s(1), s(2), s(3), \ldots \rangle$	trace notation	
$\langle\ \rangle$	empty trace	
$\#s$	length of s	
$\operatorname{ran} s$	range of s	$\operatorname{ran}\langle a, b, c, b, b \rangle = \{a, b, c\}$
$head(s)$	head of s	$head(\langle a, b, c, b, b \rangle) = a$
$front(s)$	front of s	$front(\langle a, b, c, b, b \rangle) = \langle a, b, c, b \rangle$
$tail(s)$	tail of s	$tail(\langle a, b, c, b, b \rangle) = \langle b, c, b, b \rangle$
$last(s)$	last of s	$last(\langle a, b, c, b, b \rangle) = b$
$s \leq^n u$	s prefix of u, $\#u - \#s \leq n$	$\langle a, b \rangle \leq^1 \langle a, b, c \rangle$
$s \trianglelefteq u$	s subtrace of u	$\langle a, c \rangle \trianglelefteq \langle a, b, c, d \rangle$
s **in** u	s segment of u	$\langle b, c \rangle$ **in** $\langle a, b, c, d \rangle$
$s^\frown u$	concatenation of s and u	$\langle b, c \rangle^\frown \langle a, b, c, d \rangle = \langle b, c, a, b, c, d \rangle$
$s \upharpoonright X$	s projected on elements of set V	$\langle a, b, c, d, c \rangle \upharpoonright \{a, c\} = \langle a, c, c \rangle$
$f^*(s)$	function f applied to each element of s	$f^*(\langle b, c \rangle) = \langle f(b), f(c) \rangle$
$ch^*(s)$	trace s projected on its channels	$ch^*(\langle c.x, d.y \rangle) = \langle c, d \rangle$
$val^*(s)$	trace s projected on its values	$val^*(\langle c.x, d.y \rangle) = \langle x, y \rangle$
$\{\!\mid c \mid\!\}$	set of all channel events of c	

CSP Processes

Symbol	Meaning
$\alpha(P)$	alphabet of P
$Traces(P)$	traces of P
$Fail(P)$	failures of P
$Ref(P)$	refusals of P
$refMax(P)$	maximal refusals of P
$Div(P)$	divergences of P
$[P]^0$	next events possible for process P:
	$\{e \in \alpha P \mid (\exists\, u \in Traces(P) \bullet head(u) = e)\}$
P/s	process P after having performed trace s
$STOP$	deadlock process
$SKIP$	terminating process, producing event $\surd$
$a \rightarrow P$	prefixing operator
$P \sqcap Q$	internal choice
$(\sqcap_{x:\{a_1,\ldots,a_n\}}(x \rightarrow P(x))$	abbreviates $a_1 \rightarrow P(a_1) \sqcap \ldots \sqcap a_n \rightarrow P(a_n)$
$P \mathrel{\Box} Q$	external choice
$(x : \{a_1, \ldots, a_n\} \rightarrow P(x))$	abbreviates $a_1 \rightarrow P(a_1) \mathrel{\Box} \ldots \mathrel{\Box} a_n \rightarrow P(a_n)$
$P\,;Q$	sequential composition
$X = F(X)$,	
$\mu\, X \bullet F(X)$	recursive equation
$g\&B$	guarded command, abreviates
	if g **then** B **else** $STOP$
$P \parallel Q$ parallel operator	
$P \interleave Q$	interleaving operator
$P \setminus H$	hiding operator
$P\hat{\ }Q$	interrupt operator
$(P \wr \Delta_P)$	threat introduction operator

Appendix B. A Quick-Reference Guide to CSP

The CSP semantics $[\![\, \bullet \,]\!]$ for nondeterministic processes [44] maps each syntactic CSP unit P onto a triple of sets

$$\mathcal{I}[\![\, P \,]\!] = (A, F, D)$$

subject to the following conditions:

D0: A, F, D have the following structure:

- A is a finite set, called the *alphabet* of P, denoted $A = \alpha P$.
- $F \subseteq A^* \times \mathbb{P}\, A$ is called the *failures* of P, denoted $F = \mathit{Fail}(P)$. For each element $(s, X) \in F$, s is called a *trace* of P and X a *refusal (set)* of P/s. The set of all traces is denoted by $\mathit{Traces}(P)$.
- $D \subseteq A^*$ is called the *divergences* of P, denoted $D = \mathit{Div}(P)$.

C0: $(\langle\ \rangle, \emptyset) \in F$

C1: $(s \frown t, X) \in F \Rightarrow (s, \emptyset) \in F$

C2: $(s, Y) \in F \wedge X \subseteq Y \Rightarrow (s, X) \in F$

C3: $(s, X) \in F \wedge x \in A \Rightarrow$
$(s, X \cup \{x\}) \in F \vee (s \frown \langle x \rangle, \emptyset) \in F$

C4: $D \subseteq \operatorname{dom} F$

C5: $s \in D \wedge t \in A^* \Rightarrow s \frown t \in D$

C6: $s \in D \wedge X \subseteq A \Rightarrow (s, X) \in F$

Three refinement relations between processes P, Q are defined by means of the semantics $[\![P]\!] = (A, F, D)$ and $[\![Q]\!] = (A', F', D')$:

Trace Refinement: $P \sqsubseteq_T Q \equiv_{df} A = A' \wedge \mathit{Traces}(Q) \subseteq \mathit{Traces}(P)$

Failures Refinement: $P \sqsubseteq_F Q \equiv_{df} A = A' \wedge F' \subseteq F$

Failures-Divergence Refinement: $P \sqsubseteq_F Q \equiv_{df} A = A' \wedge F' \subseteq F \wedge D' \subseteq D$

We explain the meaning of CSP operators by giving their failure sets and their divergences.

Nondeterministic OR Operator The OR operator $\sqcap$ selects one of its operands nondeterministically without giving th environment the possibility to influence this decision:

$$Fail(P \sqcap Q) = Fail(P) \cup Fail(Q)$$

$$Div(P \sqcap Q) = Div(P) \cup Div(Q)$$

Prefixing Operator The prefixing operator $\rightarrow$ executes $P(x)$ after occurrence of an initial event $x \in B$.

$$\begin{aligned} &Fail(x : B \rightarrow P(x)) = \\ &\qquad \{(\langle\,\rangle, U) \mid U \subseteq (\alpha P - B)\} \cup \{(\langle x \rangle ^\frown s, U) \mid x \in B \wedge (s, U) \in Fail(P(x))\} \end{aligned}$$

$$Div(x : B \rightarrow P(x)) = \{\langle x \rangle ^\frown s \mid s \in Div(P(x)) \wedge x \in B\}$$

Parallel Operator The parallel operator $\|$ defines joint execution of P and Q with events common to both their alphabets executed synchronously.

$$\begin{aligned} &Fail(P \parallel Q) = \\ &\qquad \{(s, U \cup V) \mid s \in (\alpha P \cup \alpha Q)^* \wedge \\ &\qquad\qquad (s \upharpoonright \alpha P, U) \in Fail(P) \wedge (s \upharpoonright \alpha Q, V) \in Fail(Q)\} \cup \\ &\qquad \{(t, U) \mid t \in Div(P \parallel Q)\} \end{aligned}$$

$$\begin{aligned} &Div(P \parallel Q) = \\ &\qquad \{s ^\frown t \mid t \in (\alpha P \cup \alpha Q)^* \wedge \\ &\qquad\qquad\qquad (s \upharpoonright \alpha P \in Div(P) \wedge s \upharpoonright \alpha Q \in Traces(Q) \vee \\ &\qquad\qquad\qquad\; s \upharpoonright \alpha Q \in Div(Q) \wedge s \upharpoonright \alpha P \in Traces(P))\} \end{aligned}$$

General Choice Operator The alternative operator $\square$ executes P if the initial event is from P's alphabet and not from Q's alphabet, executes Q if the initial event is from Q's alphabet and not from P's alphabet and choses nondeterministically if the initial event is in the intersection of their alphabets.

$$\begin{aligned} &Fail(P \,\square\, Q) = \\ &\qquad \{(s, U) \mid (s, U) \in Fail(P) \cap Fail(Q) \vee (s \neq \langle\,\rangle \wedge (s, U) \in Fail(P) \cup Fail(Q))\} \\ &\qquad \cup \{(s, U) \mid s \in Div(P \,\square\, Q)\} \end{aligned}$$

$$Div(P \,\square\, Q) = Div(P) \cup Div(Q)$$

Hiding Operator The hiding operator $\backslash$ defines a new process as the original P with all events from C hidden.

$$\begin{aligned} &Fail(P \setminus C) = \\ &\qquad \{(t \upharpoonright (\alpha P - C), U) \mid (t, U \cup C) \in Fail(P)\} \cup \{(t, U) \mid t \in Div(P \setminus C)\} \end{aligned}$$

$$\begin{aligned} &Div(P \setminus C) = \\ &\qquad \{(s \upharpoonright (\alpha P - C)) ^\frown t \mid t \in (\alpha P - C)^* \wedge \\ &\qquad\qquad (s \in Div(P) \vee (\forall\, n \bullet (\exists\, u \in C^* \bullet (\#u > n \wedge (s ^\frown u) \in Traces(P)))))\} \end{aligned}$$

Interleaving Operator The interleaving operator $|||$ defines interleaved execution of P and Q with events common to both alphabets nondeterministically executed either by P or by Q.

$$
\begin{aligned}
&\mathit{Fail}(P \mathbin{|||} Q) = \\
&\qquad \{(s, U) \mid (\exists\, t, u \bullet s \ \mathit{interleaves}(t, u) \wedge (t, U) \in \mathit{Fail}(P) \wedge (u, U) \in \mathit{Fail}(Q))\} \cup \\
&\qquad \{(s, U) \mid s \in \mathit{Div}(P \mathbin{|||} Q)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Div}(P \mathbin{|||} C) = \\
&\qquad \{u \mid (\exists\, s, t \bullet u \ \mathit{interleaves}(s, t) \wedge \\
&\qquad\qquad (s \in \mathit{Div}(P) \wedge t \in \mathit{Traces}(Q) \vee s \in \mathit{Traces}(P) \wedge t \in \mathit{Div}(Q)))\}
\end{aligned}
$$

Interrupt Operator: $(P\hat{\ }Q)$ is defined to be the process that acts like P, but may be interrupted at any time by the first event of Q. From this event on, Q continues to run and P is never resumed. We have

$$\alpha(P\hat{\ }Q) = \alpha P \cup \alpha Q$$

and

$$\mathit{Traces}(P\hat{\ }Q) = \{s \frown t \mid s \in \mathit{TracesP} \wedge t \in \mathit{TracesQ}\}$$

It is assumed that Q's initial events are not contained in αP.