Combining Tools for the Verification of Fault-Tolerant Systems

Bettina Buth, Rachel Cardell-Oliver, Jan Peleska

Abstract. In this article, we describe an approach for the tool-supported development and verification of fault-tolerant systems according to the invent&verify paradigm. Our method is based on the CSP (Communicating Sequential Processes) specification language. It allows the desired properties of a system to be expressed as implicit specifications (assertions about traces and refusals), explicit specifications (CSP process terms), refinement relations or combinations of these three description formalisms. From our experience with industrial verification projects, this possibility to choose between different specification paradigms according to the specific needs of each development step is essential to cope with large-scale formal development and verification projects. Each top-down development step according to the invent&verify paradigm introduces a verification obligation whose type depends on the specification techniques applied for the different components involved in the step. We describe several strategies optimised for discharging specific forms of these obligations typically arising during a development. Application of these strategies is supported by three verifications tools: HOL is used to develop generic theories and apply them in strategies applicable for the verification of implicit specifications. FDR is used for the verification of refinement relations between CSP processes. PAMELA/PVS is applied for the verification of implicit assertions stated for explicit sequential processes. The main objective of this article is to show how these different specification formalisms and associated verification support tools can be most effectively combined to cope with complex fair-sized developments. Our approach is illustrated by means of a case study concerning the development of a fault-tolerant dual computer system.

1 Introduction

1.1 Motivation

The formal methods developed over the last decades have been advertised as a means for improving software quality in general and for increasing confidence in safety critical software in particular. Nevertheless, the impact of formal methods on industrial software development has been minimal. Many reasons are given for this, including their unsuitability for the development of large systems (scalability), the dependence on trained staff to apply formal methods, and the lack of tool support for their application.

From our point of view there is yet another important problem, namely that although most formal methods have been developed to support a specific part of the development process, people have tried to extend their use beyond those parts, where the method may not be suitable. In recent years this observation has led to increased interest in ways to link theories and integrate related tools.

In this paper we describe our approach to solving the verification problems which arise in a development involving different forms of specification and development steps. In the subsequent sections our research is exemplified by a case study on the design of fault-tolerant computer systems first presented in [18]. The development in this example is based on the invent&verify paradigm, where the design levels are represented as CSP specifications and processes [16, 23]. In [18] a formal theory to justify the links between each different phase of the development method has been presented. Here we consider what kind of proof tool support would be appropriate for the different development steps.

1.2 The Invent&Verifiy Paradigm

Development according to the *invent&verifiy paradigm* starts by regarding the component P to be developed (initially, P is the full system SYS) as a black box. At first, the component interface $\mathcal{I}[P]$ is defined and the desired behaviour – as far as visible at $\mathcal{I}[P]$ – is described in a requirements specification $\mathcal{R}[P]$. The system development now proceeds by performing alternating *invent steps* and *verification steps* in a recursive way. An invent step comprises four activities:

1. A number of sub-components $P_1, \ldots, P_n$ is introduced for P.
2. For each sub-component P_i, the interface $\mathcal{I}[P_i]$ is defined.
3. An *architectural decomposition* $\mathcal{D}[P]$ describing the mode of interaction between the P_i (communication paths, parallel or sequential operation etc.) is specified.
4. For each sub-component P_i, a requirements specification $\mathcal{R}[P_i]$ referring to P_i's behaviour visible at $\mathcal{I}[P_i]$ is produced.

Each invent step for a component P induces a *verification obligation* $\mathcal{V}[P]$ which may be informally expressed as

> If the P_i fulfil their requirements $\mathcal{R}[P_i]$ and interact as defined in the architectural decomposition $\mathcal{D}[P]$, then P, implemented by $P_1, \ldots, P_n$ as specified in $\mathcal{D}[P]$, will satisfy its requirements $\mathcal{R}[P]$.

The obligation $\mathcal{V}[P]$ has to be discharged in a verification step. If the requirements for a component P_i is directly implementable, the development for P_i stops. In this case we will denote P_i's requirements by $\mathcal{P}[P_i]$ to indicate that they represent an implementable process or program module. Recursive application of invent and verification steps results in a *development tree* with the system SYS as root, requirements specifications, architectural decompositions combined with verification obligations as intermediate nodes and components P_i whose requirements $\mathcal{P}[P_i]$ are directly implementable as leaves. For the case study to be presented in this paper, such a tree is depicted in Figure 2 below.

Obviously, the invent&verify paradigm is generic in the sense that it may be instantiated with different methods – formal or informal – for the specification of interfaces, requirements and architectural decompositions and for the execution of verification steps.

1.3 CSP Instantiation of the Invent&Verifiy Paradigm

The CSP-based developments considered in this article start with a black-box process SYS denoting the system to be developed and stops as soon as all the CSP processes P_i involved are represented by explicit CSP processes considered to be implementable. Formally speaking, SYS initially denotes an unbound CSP process variable, and the development is finished when all process variables $SYS, P_i, \ldots$ involved are bound by explicit (possibly recursive) process terms.

Interface Specifications. The interface of a CSP process P is defined as the subset $I[P]$ of channels or single events from a universal alphabet Σ that P is willing to share with its environment.

Requirements Specifications. We allow three types of specifications to be used in a development: (1) *implicit requirements specifications* are expressions $P \setminus (\Sigma - I[P])$ **sat** $S(s, R)$ where S (the specification) is a predicate with free variables (s, R) denoting the *failures* (i. e., pairs of traces and associated refusals) of a CSP process. The requirement asserts that every failure (s, R) of P with all internal events hidden shall satisfy specification $S(s, R)$. (2) *Refinement requirements specifications* are expressions $Q \sqsubseteq_{FD} P \setminus (\Sigma - I[P])$ where Q is an explicitly defined CSP process and $\sqsubseteq_{FD}$ denotes refinement in the failures-divergence model. The requirements states that the behaviour of P as far as visible at its interface shall be a refinement of the possible behaviours of Q. (3) *Explicit requirements specifications* are process terms $P = \ldots$ without any unbound process variables. They can be considered as stronger forms of refinement relations, since semantic equivalence $P = Q$ is defined as $Q \sqsubseteq_{FD} P \wedge P \sqsubseteq_{FD} Q$.

Observe that we allow more than one specification style to be used in a requirements definition $\mathcal{R}[P]$: It is often convenient to express some properties of a process by implicit specifications while others are better characterised as refinement relations. Examples for this technique will be given in the subsequent sections.

Architectural Decompositions. For the type of developments supported by our approach, the architectural decomposition is formally specified as an expression

$$P = D(P_1, \ldots, P_n)$$

where $D(X_1, \ldots, X_n)$ is a CSP term with unbound variables X_i using only the CSP operators $\underset{I}{\|}$ (parallel composition with synchronisation over an interface

I), $|||$ (interleaving) and $\underset{J}{\nabla}$ (interrupt by events from the set J). The architectural decomposition has to provide *interface consistency*, that is, the following conditions must be satisfied by the choice of D and the interfaces $I[P_i]$:

1. Sub-component interfaces implement the interface of P: $I[P] \subseteq \bigcup_{i=1}^{n} I[P_i]$.
2. Parallel composition respects the sub-component interfaces: For each parallel operator $\underset{J}{\|}$ contained in A, $J \subseteq \bigcup_{i=1}^{n} I[P_i]$ holds.
3. Interrupt respects the sub-component interfaces: For each interrupt operator $\underset{J}{\nabla}$ contained in A, $J \subseteq \bigcup_{i=1}^{n} I[P_i]$ holds.

We assume that the operators contained in D are directly implementable in the target system, for example by using the OCCAM programming language or operating systems providing CSP-like communication mechanisms. As a consequence, $D(P_1, \ldots, P_n)$ will be implementable as soon as $P_1, \ldots, P_n$ are associated with implementable specifications[1].

1.4 Verification Obligations

Depending on the forms of requirements specifications $\mathcal{R}[P]$, $\mathcal{R}[P_i]$ used for a component P and its sub-components P_i, the following types of verification obligations may arise: (1) *Implicit-Spec Verification Obligations* are of the form

$$\begin{aligned} &\textstyle\bigwedge_{i=1}^{n} P_i \setminus (\Sigma - I[P_i]) \ \mathbf{sat}\ S_i(s,R) \wedge P = D(P_1, \ldots, P_n) \\ &\qquad\qquad \Rightarrow P \setminus (\Sigma - I[P]) \ \mathbf{sat}\ S(s,R) \end{aligned}$$

Typically, implicit-spec proof obligations occur during the higher-level decomposition steps, where a system regarded as a black box is decomposed into smaller black-boxes also specified in an implicit way.
(2) *Mixed-Spec Verification Obligations* are of the form

$$\begin{aligned} &\textstyle\bigwedge_{i=1}^{n} Q_i \sqsubseteq_{FD} P_i \setminus (\Sigma - I[P_i]) \wedge P = D(P_1, \ldots, P_n) \\ &\qquad\qquad \Rightarrow P \setminus (\Sigma - I[P]) \ \mathbf{sat}\ S(s,R) \end{aligned}$$

where the Q_i are explicit CSP process terms. Instead of a refinement relation, the P_i could also be defined as explicit process equations $P_i = Q_i$. Mixed-spec proof obligations arise naturally when an implicitly specified component is decomposed into sub-components associated with refinement or explicit specifications, that is, when black-box representations are gradually replaced by explicit ones.
(3) *Explicit-Spec Proof Obligations* are of the form

$$\begin{aligned} &\textstyle\bigwedge_{i=1}^{n} Q_i \sqsubseteq_{FD} P_i \setminus (\Sigma - I[P_i]) \wedge P = D(P_1, \ldots, P_n) \\ &\qquad\qquad \Rightarrow Q \sqsubseteq_{FD} P \setminus (\Sigma - I[P]) \end{aligned}$$

[1] Observe that the CSP process algebra provides one language for architectural decompositions and for explicit specifications. In other formalisms, as for example, the formalisms implemented in the Statemate tool [15], different languages are used to specify requirements (e. g., State- and Activity Charts) and architecture (e. g., Module Charts).

where Q and Q_i are explicit CSP process terms. Again, the relation $\sqsubseteq_{FD}$ may be exchanged by $=$. Explicit-spec proof obligations arise during the lower-level decomposition steps, when explicit processes are refined by "smaller" sets of cooperating components.

1.5 Verification Strategies

To discharge a verification obligation, we can refer directly to the properties of the underlying semantic model, apply the algebraic laws provided for explicit CSP processes or the compositional proof theory allowing to derive satisfaction relations valid for a composed process from those fulfilled by the lower-level components. However, the efficiency of the verification process for complex systems is considerably increased, if proof obligations can be discharged according to *strategies*, that is, standardised reusable "recipes" combining several steps performed in the semantic domain or the proof theory into one "macro" verification step. Application of strategies well-suited for the type of verification to be performed helps to transform the "invention" of correctness proofs into a routine matter. Moreover, it facilitates the application of tools that are optimised for specific strategies occurring naturally during the verification process.

From our experience the following strategies turned out to be most helpful for the CSP-based development of fault-tolerant systems. For each of these strategies, a verification example will be presented in the case study below.

Application of Generic Theories Generic theories represent collections of proof rules that are valid for a class of processes. The class members have certain generic behavioural properties in common and may be instantiated with concrete sets of events, channels names and constants[2]. The proof rules of a generic theory may be instantiated by these concrete entities along with the processes. Compared to verification procedures referring to concrete processes, the advantage of generic theories lies in their higher re-usability: They may be applied to all processes that follow a certain design pattern but may be distinct with respect to the sets of events, channels and constants involved. Moreover, good system design practice tries to apply similar design patterns for as many system components as possible. Therefore, the chances for reusing a generic theory that has proven helpful in one verification obligation are increased with the quality of the system design. As an example, a generic theory of generalised pipes will be applied in the case study. Further applications of other generic theories are described in [6, 9, 10, 22].

Abstraction to Finite State/Finite Value Obligations. If the verification of explicit-spec obligations can be expressed as a refinement relation between

[2] Observe that the algebraic laws and the compositional proof theory of CSP are already generic, since they may be instantiated with different sets of events etc. However, we would not call these basic laws generic theories since they do not represent macro verification steps but represent the "smallest units of reasoning".

finite state processes using finite data types and concrete data transformations only, it can be discharged by *model checking*, that is, complete exploration of the process state spaces involved. While tool-supported theorem proving requires manual interaction and expert knowledge about the proof tool, model checking can be completely mechanised. It is therefore highly attractive for practical application in an industrial context. However, the original processes involved in a refinement proof obligation $Q \sqsubseteq_{FD} P$ often have state spaces too large to be tackled by a model checking tool. In such a situation it is a promising strategy to find a *valid abstraction* A transforming P and Q to new processes $A(P)$ and $A(Q)$ with smaller state spaces, data types etc. An abstraction is valid for the refinement problem $Q \sqsubseteq_{FD} P$, if $A(Q) \sqsubseteq_{FD} A(P)$ implies $Q \sqsubseteq_{FD} P$.

Auxiliary Processes Satisfying Implicit Specifications. For the verification of mixed-spec verification obligations it is necessary to prove that an explicitly specified process P or one satisfying a certain refinement relation $Q \sqsubseteq_{FD} P$ will also satisfy an implicit assertion. Recalling that for each implicit specification there exists a most non-deterministic explicit CSP process satisfying it (c. f. [23]), the following strategy is motivated: Given the implicit specification, try to find a (preferably simple) explicit auxiliary process X, such that X **sat** $S(s, R)$ can be proved without too much effort (for example, by using the next strategy below). If P has been explicitly specified, investigate whether $X \sqsubseteq_{FD} P$ holds; if P has been specified by refinement relation $Q \sqsubseteq_{FD} P$, investigate whether $X \sqsubseteq_{FD} Q$ holds. Since refinement preserves satisfaction relations, validity of the refinement relations $X \sqsubseteq_{FD} Q, P$ implies P **sat** $S(s, R)$.

Transformation Into the Sequential Program Domain. In many cases, the auxiliary processes used according to the previous strategy can be constructed as sequential processes. Therefore it is advisable to focus on the verification of mixed-spec proof obligations for *sequential* CSP processes. A strategy for this task is to transform a sequential CSP process P into a corresponding *sequential nondeterministic program* $\nu(P)$ interpreted in an operational semantics. In [18] it has been proven that validity of the implicit assertion P **sat** $S(s, R)$ is equivalent to $\nu(P)$ satisfying a specific program invariant. This allows us to apply Hoare-style reasoning and tools supporting this verification style in the sequential program domain and apply the results achieved to implicit assertions about CSP processes.

Decomposition of Proof Obligations. Observe that in order to discharge one verification obligation, it may be advisable to first decompose the obligation into a collection of equivalent "smaller" obligations that may be verified using one of the strategies above. A typical example for this approach is the decomposition into an obligation that may be abstracted to a finite state/finite value obligation and another one referring to generic data types and functions or/and to unbounded sets of process states. This technique is demonstrated in the case study below.

1.6 Tool Support

For the efficient application of the strategies sketched above, we have selected three tools that are very well-suited for the specific verification steps involved:

HOL With CSP Encoding. HOL is a general purpose theorem prover allowing interactive reasoning in higher-order logic [14]. For our purposes we have used an encoding of CSP [8] in HOL to develop a generic theory of generalised pipes for the case study. The full encoding is described in [7], the case study below will show the major theorems and HOL encodings of verification obligations. The utilisation of HOL in our context is twofold: First, we use the tool to develop and verify new generic theories with the objective to build a library of theories useful in several verification projects. Second, HOL is used to perform compositional reasoning using the instantiated theories and general compositional reasoning to discharge the implicit-spec verification obligations occurring during a development.

FDR. The Failures-Divergence Refinement model checker FDR of Formal Systems Ltd [12] is used for the verification of explicit finite state, finite value refinement relations arising from explicit-spec and mixed-spec proof obligations. It performs fully automatic refinement proofs by means of model checking of the transition graphs representing the processes involved.

PAMELA/PVS. To perform invariant proofs for nondeterministic sequential programs according to the strategy described above, the PAMELA/PVS tool is used [4, 5, 24]: PAMELA is a front-end tool supporting VDM-like specifications and specifications about nondeterministic sequential programs. It generates the relevant proof obligations and uses PVS as a back-end for discharging them. PVS is currently one of the most advanced specification and verification systems for software applications. It's specification language is higher order logic and it provides a number of verification and support tools.

2 A Fault-Tolerant Dual Computer System – System Description

To illustrate our approach, we describe the development and verification of a fault-tolerant dual computer system. Since this article focuses more on the aspects of verification and associated tool support than on fault-tolerance, the case study is slightly simplified: It only describes situations where the master computer fails and does not handle re-integration, so that only one failure is considered. The full system whose specification is completely symmetric in both computers and also handles re-integration of repaired components is described in [18, pp. 56].

The objective of the case study is to develop a fault-tolerant server system SYS for the client-server configuration, whose complete architectural decomposition is depicted in Figure 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.

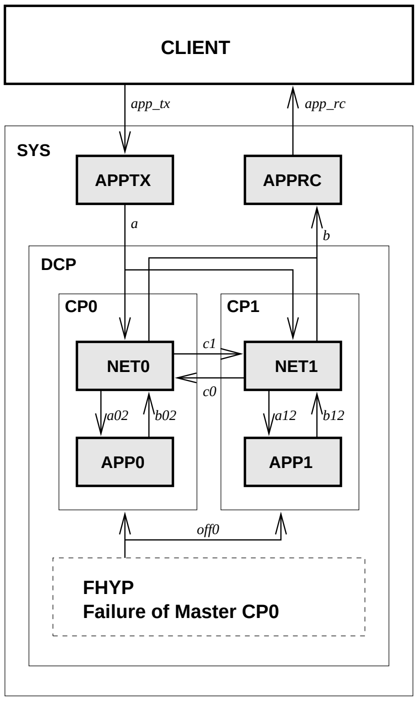

Figure 1. Architectural decomposition of the fault-tolerant server system.

The server SYS shall be designed as a dual-computer system operating in master-standby mode. As long as at least one of the computers is operable (below we will only consider the failure of the master computer), the following functionality shall be guaranteed:

1. SYS returns results in the order of the corresponding inputs.
2. 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[3].

Before presenting the systematic top-down development according to the invent&verify paradigm in the subsequent sections, it is helpful to sketch the general idea of the fault-tolerance concept. The application layer of the client computer is supported by a network layer consisting of processes $APPTX$ and $APPRC$ implementing a filter by means of an alternating bit protocol. Process $APPTX$ associates an alternating bit b with every client job x received on *app_tx* and broadcasts the resulting message (x, b) on channel a to the dual computer DCP, where it is received by both CP_0 and CP_1.

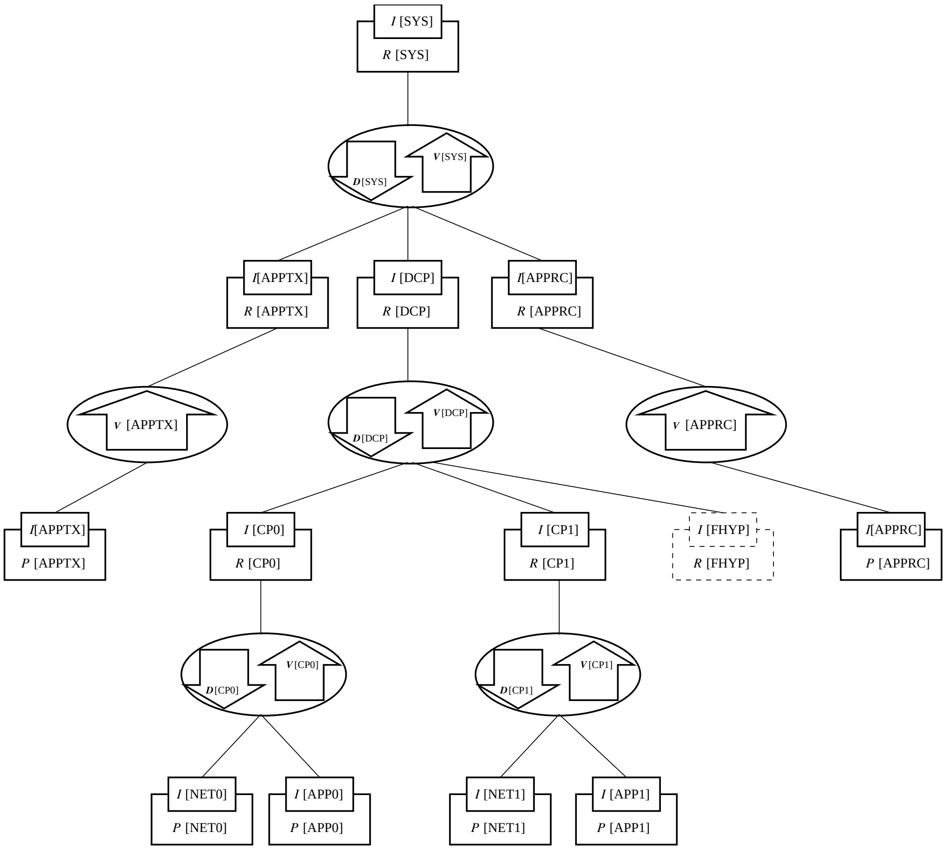

Figure 2. Development tree for fault-tolerant system SYS.

As long as both computers are operable, the master CP_0 performs the compu-

3 Note that this implies non-divergence of SYS and is a stronger requirement than the predicate $NODIV$ introduced in [16, p. 125], stating that the internal events are bounded by a function of the number of visible events already produced.

tation $y := f(x)$ and returns the result together with the identification bit b on channel b. The standby computer CP_1 stores the original input (x, b), until it receives an acknowledge on channel c_1. If the master computer fails, the failure event off_0 will also be sensed by the standby CP_1, who then takes over and processes the jobs received from the client. If CP_0 failed while still working on job (x, b), this is detected by CP_1 since the corresponding acknowledge on c_1 is still missing. In this case CP_1 will redo this job in place of CP_0 and return the result $(f(x), b)$ to the client. If the failure of CP_0 occurred *after* having delivered the computation result on b and *before* sending the acknowledge on c_1 to CP_1, this will lead to a duplicated computation $(f(x), b)$ arriving at the filter $APPRC$. The duplication is detected by this process by evaluation of the bit b, and the superfluous message is not passed on to the client application via channel $\mathit{app\_rc}$ but discarded instead.

In the sections to follow, the systematic development of SYS following the invent&verify paradigm will be presented. Figure 2 presents a *development tree* arranging the interface definitions, requirements specifications, architectural decompositions and verification obligations created during the development in top-down fashion.

3 Data Dictionary

In this section, the data types, constants, channels and generic specifications used in subsequent development steps are defined. These definitions may be regarded as the *data dictionary* of our development, where all relevant definitions are maintained in a central database.

TYPE DEFINITIONS:

$[DATA]$
$BIT == \{0, 1\}$

Type $DATA$ is unspecified, but assumed to be finite; the development below will be generic in $DATA$ and in the data transformation function

$$f : DATA \to DATA$$

applied by the computation server SYS.

$$id : DATA \to DATA$$

denotes the identity function. In addition, we will use an auxiliary function

$$\rho : (DATA \times BIT)^* \to (DATA \times BIT)^*$$

operating as a filter function on sequences of $(data, bit)$-pairs by deleting entries containing the same bit-value as their predecessor. ρ is recursively defined by

$$\rho(\langle\ \rangle) =_{df} \langle\ \rangle$$
$$\rho(s \frown \langle(x, b)\rangle) =_{df} \textbf{if } \#s > 0 \wedge bit(last(s)) = b \textbf{ then } \rho(s) \textbf{ else } \rho(s) \frown \langle(x, b)\rangle$$

Function

$$data : (DATA \times BIT) \rightarrow DATA$$

denotes the projection of a pair (x, b) on its data component x, function

$$bit : (DATA \times BIT) \rightarrow BIT$$

the projection on b.

CHANNEL DEFINITIONS:

$channel\ app\_tx : DATA$
$channel\ app\_rc : DATA$
$channel\ a : DATA \times BIT$
$channel\ b : DATA \times BIT$
$channel\ a_{02} : DATA$
$channel\ b_{02} : DATA$
$channel\ a_{12} : DATA$
$channel\ b_{12} : DATA$
$channel\ off_0$
$channel\ c_0$
$channel\ c_1$

In these definitions, channels without associated data type are considered as unstructured signals. We use auxiliary functions val and ch generically defined on channel events $c.x$ by $val(c.x) =_{df} x$ and $ch(c.x) =_{df} c$.

For any function $g : X \rightarrow Y$ we use notation $g^* : X^* \rightarrow Y^*$ for the lifting of g to sequences over X and Y; $g^*(\langle x_1, \ldots x_n\rangle) =_{df} \langle g(x_1), \ldots g(x_n)\rangle$.

It is useful to introduce some implicit specifications about traces and refusals that are generic with respect to channels, data transformation functions and bounding functions. It will turn out that several processes to be introduced during the development will satisfy instances of these specifications.

Specification NO_BLOCK is generic in the list of input channels in_i and output channels out_j and states that either all inputs or at least one output are accepted, which is the essential non-blocking property of pipes:

$$NO\_BLOCK[(in_1, \ldots, in_n), (out_1, \ldots, out_m)](s, R) \equiv_{df}$$
$$\{\!|\ in_1, \ldots, in_n\ |\!\} \cap R = \emptyset \vee \{\!|\ out_1, \ldots, out_m\ |\!\} \not\subseteq R$$

Notation $\{\!|\ c\ |\!\}$ is used for the set of all events $c.x$ where x is an element of the channel alphabet $\alpha(c)$.

Specification *LEFT_GUARD* states the safety property that the number of output events contained in a trace is bounded by an $\mathbb{N}$-valued function β of the trace restricted to its inputs.

$$\begin{array}{l} \mathit{LEFT\_GUARD}[(in_1,\ldots,in_n),(out_1,\ldots,out_m),\beta](s) \equiv_{df} \\ \qquad\qquad \#(s \upharpoonright \{\!|\ out_1,\ldots,out_m\ |\!\}) \leq \beta(s \upharpoonright \{\!|\ in_1,\ldots,in_n\ |\!\}) \end{array}$$

As bounding function $\beta : \Sigma^* \to \mathbb{N}$ we will mostly use $n \cdot \#(\cdot)$, that is, the length of a trace multiplied by a factor $n \in \mathbb{N}$.

The *COPY*-specification describes how inputs are mapped to outputs. This is expressed by defining functions $f_1 : A^* \to A^*$ and $f_2 : A^* \to A^*$, so that application of f_2 to the sequence of output values in the trace equals a prefix of the sequence of the input values transformed by f_1. (The set A denotes the union of all channel alphabets, that is, the functions f_i are applied to the values that may be passed along input and output channels.)

$$\begin{array}{l} \mathit{COPY}[(in_1,\ldots,in_n),f_1,(out_1,\ldots,out_m),f_2,N](s) \equiv_{df} \\ \qquad f_2(val^*(s \upharpoonright \{\!|\ out_1,\ldots,out_m\ |\!\})) \leq^N f_1(val^*(s \upharpoonright \{\!|\ in_1,\ldots,in_n\ |\!\})) \end{array}$$

4 Top-Level Requirements

The requirements for the server *SYS* can be formalised by the system interface definition

INTERFACE DEFINITION $\mathcal{I}[SYS]$:

DEFINITION:	$I(SYS) =_{df} \{\!|\ app\_tx, app\_rc\ |\!\}$
INPUTS:	app_tx
OUTPUTS:	app_rc

and the implicit specification

REQUIREMENTS SPECIFICATION $\mathcal{R}[SYS]$:

IMPORT: $\mathcal{I}[SYS]$
IMPLICIT REQUIREMENT: $SYS \setminus (\Sigma - I(SYS))$ **sat** $S_{SYS}(s, R)$
IMPLICIT SPECIFICATION:

$$\begin{array}{l} S_{SYS}(s,R) \equiv_{df} \\ \qquad \mathit{NO\_BLOCK}[(app\_tx),(app\_rc)](s,R) \\ \qquad\qquad \wedge\ \mathit{COPY}[(app\_tx),f^*,(app\_rc),id^*,N](s) \end{array}$$

The pair $(\mathcal{I}[SYS], \mathcal{R}[SYS])$ is depicted as the root of the development tree shown in Figure 2. Functions f, id and the channels referenced are defined in the data dictionary above. $N = 3$ is a constant fixing the buffering depth allowed for the server. $\mathcal{R}[SYS]$ requires that SYS shall never block communication on both interface channels at the same time and that the results returned on the output channel are a prefix of the sequence resulting from application of f to each input value.

5 Invent&Verify Steps for SYS

5.1 Decomposition of SYS Into DCP, $APPTX$ and $APPRC$

The first invent step leads us to the introduction new process variables DCP, $APPTX$ and $APPRC$. These variables are free at the present state of the development, they will be bound to concrete processes in subsequent steps. The "black box" denoted by DCP will be decomposed into the dual computer system. The other two processes will act as alternating bit filter processes. These components communicate over channels a and b that carry both the data and alternating bits.

Interface Definitions. To formalise this architectural decomposition, we first define interfaces for the new components (see Figures 1 and 2). The interface for $APPTX$ is given by

INTERFACE DEFINITION $\mathcal{I}[APPTX]$:

DEFINITION:	$I(APPTX) =_{df} \{\| app\_tx, a \|\}$
INPUTS:	app_tx
OUTPUTS:	a

The interface for $APPRC$ is given by

INTERFACE DEFINITION $\mathcal{I}[APPRC]$:

DEFINITION:	$I(APPRC) =_{df} \{\| app\_rc, b \|\}$
INPUTS:	b
OUTPUTS:	app_rc

The interface for DCP is given by

INTERFACE DEFINITION $\mathcal{I}[DCP]$:

DEFINITION:	$I(DCP) =_{df} \{\| a, b \|\}$
INPUTS:	a
OUTPUTS:	b

Architectural Decomposition. Referring to the new process variables and interfaces defined above, the architectural decomposition of SYS into the new components is specified by

ARCHITECTURAL DECOMPOSITION $\mathcal{D}[SYS]$:

IMPORT: $\mathcal{I}(SYS), \mathcal{I}(APPTX), \mathcal{I}(APPRC), \mathcal{I}(DCP)$
DECOMPOSITION:

$$SYS = D_{SYS}(APPTX, DCP, APPRC)$$
$$D_{SYS}(APPTX, DCP, APPRC) =_{df} (APPTX \underset{a}{\parallel} (DCP \underset{b}{\parallel} APPRC))$$

Requirements Specifications. The invent step is completed by describing the specifications to be fulfilled by the three processes, as far as visible at their interfaces. Process $APPTX$ inputs on channel app_tx and outputs the input data associated with alternating bits on a. It never blocks both inputs and outputs; moreover, the number of outputs is guarded by the number of inputs received.

REQUIREMENTS SPECIFICATION $\mathcal{R}[APPTX]$:

IMPORT: $\mathcal{I}[APPTX]$
IMPLICIT REQUIREMENT: $APPTX \setminus (\Sigma - I(APPTX))$ **sat** $S_{TX}(s, R)$
IMPLICIT SPECIFICATION:

$$\begin{aligned} &S_{TX}(s, R) \equiv_{df} \\ &\quad NO\_BLOCK[(app\_tx), (a)](s, R) \\ &\qquad\qquad \wedge LEFT\_GUARD[(app\_tx), (a), \#(\cdot)](s) \\ &\qquad\qquad \wedge COPY[(app\_tx), f^*, (a), f^* \circ data^* \circ \rho, 1](s) \end{aligned}$$

Observe that the computation function f only appears in the $COPY$ specification to facilitate the derivation of the composed functionality of the three processes, that is, the property $COPY[(app\_tx), f^*, (app\_rc), id^*, N](s)$. The alternating-bit property is implicitly expressed by the function ρ defined above: Since $APPTX$ never blocks at its interface and the length of the output trace filtered by ρ must not be shorter than the length of the input trace minus 1, the process has to associate alternating bits, because otherwise $COPY[(app\_tx), f^*, (a), f^* \circ data^* \circ \rho, 1](s)$ cannot be satisfied.

The dual computer system DCP has the interface behaviour of a non-blocking pipe which transforms input data by f. The output messages may be replicated, but at most once for each message. (The lower design levels will strengthen this property such that messages will only be duplicated if the master computer fails after having delivered a job and before having sent the acknowledgement to the standby computer.)

REQUIREMENTS SPECIFICATION $\mathcal{R}[DCP]$:

IMPORT: $\mathcal{I}[DCP]$
IMPLICIT REQUIREMENT: $DCP \setminus (\Sigma - I(DCP))$ **sat** $S_{DCP}(s, R)$
IMPLICIT SPECIFICATION:

$$
\begin{aligned}
& S_{DCP}(s, R) \equiv_{df} \\
& \quad NO\_BLOCK[(a), (b)](s, R) \\
& \qquad \wedge\ LEFT\_GUARD[(a), (b), 2 \cdot \#(\cdot)](s) \\
& \qquad \wedge\ COPY[(a), f^* \circ data^* \circ \rho, (b), data^* \circ \rho, 1](s)
\end{aligned}
$$

The filter process $APPRC$ has the analogous non-blocking and left-guardedness properties as the others. The $COPY$-specification states that the sequence of output values shall be a 1-prefix of the projection to the $DATA$-component of the input sequence filtered by ρ.

REQUIREMENTS SPECIFICATION $\mathcal{R}[APPRC]$:

IMPORT: $\mathcal{I}[APPRC]$
IMPLICIT REQUIREMENT: $APPRC \setminus (\Sigma - I(APPRC))$ **sat** $S_{RC}(s, R)$
IMPLICIT SPECIFICATION:

$$
\begin{aligned}
& S_{RC}(s, R) \equiv_{df} \\
& \quad NO\_BLOCK[(b), (app\_rc)](s, R) \\
& \qquad \wedge\ LEFT\_GUARD[(b), (app\_rc), \#(\cdot)](s) \\
& \qquad \wedge\ COPY[(b), data^* \circ \rho, (app\_rc), id^*, 1](s)
\end{aligned}
$$

5.2 Verification of the SYS-Decomposition: Generic Theories

The invent-step performed in the decomposition of SYS above induces the following proof obligation:

VERIFICATION OBLIGATION $\mathcal{V}[SYS]$:

IMPORT: $\mathcal{R}[APPTX], \mathcal{R}[DCP], \mathcal{R}[APPRC], \mathcal{R}[SYS], \mathcal{D}[SYS]$

HIGHER-LEVEL REQUIREMENTS:
$\mathcal{R}[SYS]$
IMPLIED BY:
lower-level requirements:
$\mathcal{R}[APPTX], \mathcal{R}[DCP], \mathcal{R}[APPRC]$
architectural decomposition:
$\mathcal{D}[SYS]$

To discharge this verification obligation, we proceed by performing the steps (1) Strategy Selection, (2) Theorem Selection, (3) Theory Instantiation and (4) Verification; this will be illustrated in the following paragraphs.

Strategy Selection. Observe that the architectural decomposition $SYS = D_{SYS}(APPTX, DCP, APPRC)$, when restricted to the interfaces $I(APPTX)$, $I(DCP)$, $I(APPRC)$ defines just a composition of pipes, that is,

$$SYS \setminus (\Sigma - I(SYS)) = APPTX \gg DCP \gg APPRC$$

where the pipe operator is defined for pipes P with channels in, mid and Q with channels mid, out as $P \gg Q =_{df} (P \underset{mid}{\|} Q) \setminus (\Sigma - \{\!|\ in, out\ |\!\})$. As a consequence it is a promising strategy to apply a *generic (generalised) pipe theory* for discharging $\mathcal{V}[SYS]$. In the context of generic theories, the HOL tool is applied for two objectives:

– Development of new generic theories,
– application of instantiated theories in concrete proofs.

The generic pipe theory to be applied in verification obligation $\mathcal{V}[SYS]$ generalises the existing pipe theory contained in [16] by admitting several input and output channels per pipe. It has been completely verified using the HOL-tool (details are described in the technical report [7]). Another application of the theory in an industrial verification project is described in [6].

Theorem Selection. Having decided on which generic theory to use, the next step consists in the selection of appropriate generic theorems provided by the theory. For the verification of $\mathcal{V}[SYS]$, two theorems from the pipe theory are of particular interest:

The `PIPE_COPY` theorem allows to derive $COPY[\ldots](s)$-properties for composed pipes $P \gg Q$, where the generic $COPY[\ldots](s)$-predicate is defined in the data dictionary above:

```
PIPE_COPY =
⊢
   ∀(left:chan list) mid right. ∀(P:buffer) Q. ∀f:trace->num.
   ∀(f1:(alpha list)->(alpha list)) f2 f3. ∀(N:num) M.
  ((SAFE_PIPE P Q left mid right f) ∧
   ((P left mid)  trsat (COPY left f1 mid   f2 N)) ∧
   ((Q mid right) trsat (COPY mid  f2 right f3 M)))
   ⇒
  (pipe P Q left mid right) trsat (COPY left f1 right f3 (N+M))
```

The theorem is presented in the syntax of the HOL encoding for CSP and the pipe theory. `trsat` denotes satisfaction of trace specifications, P **sat** $S(h)$. `(COPY mid f2 right f3 M)` stands for $COPY[mid, f2, right, f3, M](h)$. The theorem explains the input-output behaviour of composed pipes by deriving the $COPY$ specification for $P \gg Q$ from the $COPY$ specifications of the isolated processes. For applicability of the theory it is necessary that P and Q should satisfy some well-formedness conditions that guarantee non-blocking and non-divergence of $P \gg Q$. These are summarised in the `SAFE_PIPE` predicate and require that (1) output channels must not feed back into input channels, (2) the isolated pipes are non-blocking and (3) P must be left-guarded.

Another theorem provided by the pipe theory is concerned with the proof of $NO\_BLOCK[\ldots](s, R)$-properties:

```
SAFE_PIPE_NO_BLOCK

  ⊢ ∀in mid out P Q f.
      SAFE_PIPE P Q in mid out f ⇒
      (pipe P Q in mid out) sat (NO_BLOCK in out)
```

Theorem `SAFE_PIPE_NO_BLOCK` states that the non-blocking property for a composition of pipes $P \gg Q$ can be derived from the isolated pipes P and Q, if these satisfy the `SAFE_PIPE` condition.

Theory Instantiation. Next, we instantiate the theorems selected above for the different situations where they should be applied. This results in a series of lemmas stated for the concrete process names, channels and constants involved. As an example, we instantiate the `PIPE_COPY` theorem for the composed pipe $APPTX \gg DCP$:

Lemma 5.1. *If*

1. *$APPTX$ and DCP satisfy the* `SAFE_PIPE` *condition for channels* (app_tx, a, b),
2. $APPTX \setminus (\Sigma - I(APPTX))$ **sat** $COPY[(app\_tx), f^*, (a), f^* \circ data^* \circ \rho, 1](s)$,
3. $DCP \setminus (\Sigma - I(DCP))$ **sat** $COPY[(a), f^* \circ data^* \circ \rho, (b), data^* \circ \rho, 1](s)$,

then $APPTX \gg DCP$ **sat** $COPY[(app\_tx), f^*, (b), f^* \circ data^* \circ \rho, 2](s)$ *follows.*

□

The SAFE_PIPE_NO_BLOCK instantiated for $APPTX \gg DCP$ leads to

Lemma 5.2. *If $APPTX$ and DCP satisfy the* SAFE_PIPE *condition for channels (app_tx, a, b), then $APPTX \gg DCP$* **sat** *$NO\_BLOCK[(app\_tx), (b)](s, R)$ follows.* □

Similar lemmas derived from PIPE_COPY and SAFE_PIPE_NO_BLOCK can be stated for the composed pipe $(APPTX \gg DCP) \gg APPRC$.

Verification. The verification obligation $\mathcal{V}[SYS]$ is mechanically discharged by applying the lemmas above in the HOL framework. In our encoding of the generic pipe theory, $\mathcal{V}[SYS]$ is represented as a HOL theorem:

```
SYS_CORRECT =

⊢ (BUFFER_ALPHA APPTX [`apptx`] [`a`]) ∧
   (BUFFER_ALPHA DCP [`a`] [`b`]) ∧
   (BUFFER_ALPHA APPRC [`b`] [`apprc`]) ∧

   ((APPTX [`apptx`] [`a`]) sat (NO_BLOCK [`apptx`] [`a`])) ∧
   ((DCP [`a`] [`b`])       sat (NO_BLOCK [`a`] [`b`])) ∧
   ((APPRC [`b`] [`apprc`]) sat (NO_BLOCK [`b`] [`apprc`])) ∧

   ((APPTX [`apptx`] [`a`]) sat (LEFT_GUARD [`apptx`] [`a`] LENGTH)) ∧
   ((DCP [`a`] [`b`])       sat (LEFT_GUARD [`a`] [`b`] du)) ∧
   ((APPRC [`b`] [`apprc`]) sat (LEFT_GUARD [`b`] [`apprc`] LENGTH)) ∧

   ((APPTX [`apptx`] [`a`]) trsat
    (COPY [`apptx`] (MAP f_data)
          [`a`] ((MAP f_data) o (MAP m_to_d) o rho) 1)) ∧
   ((APPTX [`apptx`] [`a`]) trsat
    (COPY [`apptx`] ab [`a`] I 1)) ∧
   ((DCP [`a`] [`b`]) trsat
    (COPY [`a`] ((MAP f_data) o (MAP m_to_d) o rho)
          [`b`] ((MAP m_to_d) o rho) 1)) ∧
    ((APPRC [`b`] [`apprc`]) trsat
     (COPY [`b`] ((MAP m_to_d) o rho) [`apprc`] I 1))

   ⇒

  ((SYS_ARCH sat (NO_BLOCK [`apptx`] [`apprc`])) ∧

   (SYS_ARCH trsat (COPY [`apptx`] (MAP f_data) [`apprc`] I (1+1+1))))
```

In this HOL theorem, the data transformation function f^* is encoded as (MAP f_data), function $data^*$ as (MAP m_to_d) and the identity mapping id^* on sequences is denoted by I. SYS_ARCH denotes the HOL encoding of the architectural decomposition function D_{SYS} defined in $\mathcal{D}[SYS]$:

```
⊢def SYS_ARCH =
(pipe APPTX
      (pipe DCP APPRC [‘a‘] [‘b‘] [‘apprc‘])
      [‘apptx‘] [‘a‘] [‘apprc‘])
```

The predicates `BUFFER_ALPHA...` are well-formedness conditions on buffer processes that are trivially satisfied for $APPTX$, DCP and $APPRC$.

The HOL proof of $\mathcal{V}[SYS]$ is carried out along the following lines: At first it is proved that the processes involved satisfy the `SAFE_PIPE` conditions required in the premise of the lemmas. This follows immediately from the NO_BLOCK and $LEFT\_GUARD$ properties stated for the three processes in their requirements specifications $\mathcal{R}[APPTX]$, $\mathcal{R}[DCP]$ and $\mathcal{R}[APPRC]$.

Next, we can compose the three $COPY$ specifications recorded in $\mathcal{R}(APPTX)$, $\mathcal{R}(DCP)$ and $\mathcal{R}(APPRC)$ according to the implications of Lemma 5.1 and the corresponding lemma for $(APPTX \gg DCP) \gg APPRC$. This results in

$$APPTX \gg DCP \gg APPRC \ \textbf{sat} \ COPY[(app\_tx), f^*, (app\_rc), id^*, 3](h)$$

Finally, we evaluate the implications of Lemma 5.2 and the corresponding lemma for $(APPTX \gg DCP) \gg APPRC$ and get the required non-blocking property $NO\_BLOCK[(app\_tx), (app\_rc)](s, R)$, which completes the proof of $\mathcal{V}[SYS]$. □

6 Invent&Verify Step for $APPRC$

6.1 Implementable Process Specification of APPRC

For $APPRC$, no further architectural decomposition is necessary; we invent an explicit implementable representation as a communicating sequential process:

IMPLEMENTABLE PROCESS SPECIFICATION $\mathcal{P}[APPRC]$:

IMPORT: $\mathcal{I}[APPRC]$

EXPLICIT SPECIFICATION:

$$\begin{aligned} &APPRC = b?(y,d) \to app\_rc!y \to RC(1-d) \\ &RC(e) = b?(y,d) \to \\ &\qquad\qquad \textbf{if } d = e \textbf{ then } app\_rc!y \to RC(1-e) \textbf{ else } RC(e) \end{aligned}$$

$APPRC$ is implemented as a process which filters incoming messages (y, d) on channel b by discarding messages carrying the same bit as the preceding ones. The data component of accepted messages is delivered to the application layer via channel app_rc. Observe that $\mathcal{P}[APPRC]$ binds the process variable $APPRC$

by means of a (recursive) process equation. This variable binding is the formal indicator for the fact that no further decompositions are necessary for $APPRC$. As a consequence, $\mathcal{P}[APPRC]$ is represented as a leaf in the development tree shown in Figure 2.

6.2 Verification of APPRC

The implementable process representation defined in $\mathcal{P}[APPRC]$ has to satisfy the requirements $\mathcal{R}[APPRC]$; this induces the

VERIFICATION OBLIGATION $\mathcal{V}[APPRC]$:
IMPORT: $\mathcal{R}[APPRC], \mathcal{P}[APPRC]$
HIGHER-LEVEL REQUIREMENTS:
$\mathcal{R}[APPRC]$
IMPLIED BY:
implementable process specification:
$\mathcal{P}[APPRC]$

Strategy Selection. Since $\mathcal{P}[APPRC]$ specifies $APPRC$ as a sequential process, we apply the strategy to map $APPRC$ into its associated nondeterministic sequential program $\nu(APPRC)$ and verify the requirement $APPRC$ **sat** $S_{RC}(s, R)$ stated above in $\mathcal{R}[APPRC]$ by performing the corresponding invariant proof on $\nu(APPRC)$ **inv** $S_{RC}(s_v, R_v)$. This process will be illustrated in the following steps.

Transformation of APPRC Into Normal Form. The pre-requisite for the transformation into the sequential program domain is a specific syntactic representation of the sequential CSP process, called *(second) normal form* according to Apt and Olderog [1, 3]. A CSP process in normal form has syntactic structure

$$\begin{array}{l} P = Q(v_1, \ldots, v_n) \\ Q(x_1, \ldots, x_n) = \\ \qquad \Box_j\, g_j(x_1, \ldots, x_n) \& c_j \rightarrow Q(f_{1j}(x_1, \ldots, x_n), \ldots, f_{nj}(x_1, \ldots, x_n)) \end{array}$$

where the g_j are communication guards depending on the parameters x_i and the c_j are inputs $d!h(x_1, \ldots, x_n)$ or outputs $e?x_{i_j}$ on channels d, e of P. After such a channel communication, the process enters the next recursion with parameters changed according to functions f_{ij}. The standard procedure for normal form transformations preserving the process semantics is to introduce an auxiliary process parameter *state* identifying the syntactic location of a communication command in the original process and using this parameter to construct communication guards g_j. For $APPRC$, this procedure leads to the transformed process

$$\mathcal{N}(APPRC) = NAPPRC(0,0,0)$$

$$\begin{array}{l}
NAPPRC(state, y, e) = \\
\quad (state = 0)\& b?(y,d) \rightarrow NAPPRC(1, y, d) \\
\quad \Box \\
\quad (state = 1)\& app_rc!y \rightarrow NAPPRC(2, y, 1-e) \\
\quad \Box \\
\quad (state = 2)\& b?(y,d) \rightarrow \\
\qquad (\mathbf{if}\ d = e\ \mathbf{then}\ NAPPRC(3, y, e)\ \mathbf{else}\ NAPPRC(2, y, e)) \\
\quad \Box \\
\quad (state = 3)\& app_rc!y \rightarrow NAPPRC(2, y, 1-e)
\end{array}$$

Transformation Into the Sequential Program Domain. The transformation into the nondeterministic sequential program $\nu(APPRC)$ takes $\mathcal{N}(APPRC)$, replaces communications by variable assignments and introduces the auxiliary variables s_v and R_v modelling the trace performed so far and the refusal valid in the actual state. This results in the program

$$\begin{array}{l}
\nu(APPRC) = state := 0;\ s_v := \langle\,\rangle;\ R_v := \{\!|\ app_rc\ |\!\};\\
\qquad \mathbf{do}\\
\qquad\quad \Box_{(z,w):DATA \times BIT}((state = 0) \rightarrow\\
\qquad\qquad\qquad (y,d) := (z,w);\ state := 1;\\
\qquad\qquad\qquad s_v := s_v \frown \langle b.y.d \rangle;\\
\qquad\qquad\qquad R_v := \{\!|\ b\ |\!\};\ )\\
\qquad\quad \Box\\
\qquad\quad (state = 1) \rightarrow\\
\qquad\qquad\qquad z := y;\ e := 1 - d;\ state := 2;\\
\qquad\qquad\qquad s_v := s_v \frown \langle app_rc.y \rangle;\\
\qquad\qquad\qquad R_v := \{\!|\ app_tx\ |\!\};\\
\qquad\quad \Box\\
\qquad\quad \Box_{(z,w):DATA \times BIT}((state = 2) \rightarrow\\
\qquad\qquad\qquad (y,d) := (z,w);\\
\qquad\qquad\qquad \mathbf{if}\ d = e\ \mathbf{then}\ state := 3;\ \mathbf{else}\ SKIP;\\
\qquad\qquad\qquad s_v := s_v \frown \langle b.y.d \rangle;\\
\qquad\qquad\qquad R_v := \mathbf{if}\ state = 3\ \mathbf{then}\ \{\!|\ b\ |\!\}\ \mathbf{else}\ \{\!|\ app_tx\ |\!\};\ )\\
\qquad\quad \Box\\
\qquad\quad (state = 3) \rightarrow\\
\qquad\qquad\qquad z := y;\ e := 1 - d;\ state := 2;\\
\qquad\qquad\qquad s_v := s_v \frown \langle app_rc.y \rangle;\\
\qquad\qquad\qquad R_v := \{\!|\ app_tx\ |\!\};\\
\qquad \mathbf{od}
\end{array}$$

Verification Using PAMELA/PVS. The proof obligation for $APPRC$ can now be discharged in the sequential program domain interpreted in the operational semantics defined by Apt and Olderog [3]: In [18, p. 99, Theorem 6] we have shown that the CSP process requirement $APPRC$ **sat** $S_{RC}(s, R)$ is equivalent to the requirement

$$\nu(APPRC) \ \mathbf{inv} \ S_{RC}(s_v, R_v)$$

on the non-deterministic program $\nu(APPRC)$. This requirement states that predicate $S_{RC}(s_v, R_v)$ is an invariant of the **do** ... **od**-loop.

Verification of sequential programs based on Hoare-style axiomatisations using pre- and post conditions can be performed using the PAMELA/PVS tool. The mechanised verification of the invariant does not show any errors.

7 Invent&Verify Step for $APPTX$

7.1 Explicit Specification of APPTX

In analogy to $APPRC$, $APPTX$ is defined as a sequential process:

IMPLEMENTABLE PROCESS SPECIFICATION $\mathcal{P}[APPTX]$:

IMPORT: $\mathcal{I}[APPTX]$

EXPLICIT SPECIFICATION:

$$APPTX = TX(0)$$
$$TX(e) = app\_tx?x \rightarrow a!(x, e) \rightarrow TX(1 - e)$$

Process $APPTX$ will be implemented as a process which adds alternating bits e, starting with $e = 0$ to inputs x received on channel app_tx and outputs the pairs (x, e) on channel a. We skip the verification of the proof obligation $APPTX$ **sat** $S_{TX}(s, R)$, because this is performed in analogy to the verification of $APPRC$.

8 Invent&Verify Step for DCP

8.1 Decomposition of DCP Into Dual Computers CP_0 and CP_1

As stated in the informal system description above, component DCP shall be designed as a fault-tolerant sub-system using a dual computer technique. The invent step performed in this section deals with the introduction of the dual computers and the hypotheses about their exceptional behaviour. The dual computers are represented by the free process variables CP_0 and CP_1, the assumptions about possible system failures will be explicitly modelled by a process $FHYP$.

Interface Definitions. Both computers receive inputs on channel a in a synchronous way and output independently on channel b (see architectural decomposition $\mathcal{D}[DCP]$ below). In addition, they exchange control signals on channels c_0, c_1 and react on control signal off_0 which models the crash of computer CP_0. This leads to symmetric interface definitions $\mathcal{I}[CP_i], i = 0, 1$:

INTERFACE DEFINITION $\mathcal{I}[CP_i]$:

DEFINITION:	$I(CP_i) =_{df} \{\| a, b, c_0, c_1, \mathit{off}_0 \|\}$
INPUTS:	a, c_i, off_0
OUTPUTS:	$b, c_{(1-i)}$

In our simplified example, the fault hypotheses modelled in CSP process $FHYP$ use a single event (in the full example described in [18], the fault hypotheses synchronise a several failure and communication events):

INTERFACE DEFINITION $\mathcal{I}[FHYP]$:

DEFINITION:	$I(FHYP) =_{df} \{\| \mathit{off}_0 \|\}$
INPUTS:	
OUTPUTS:	off_0

Architectural Decomposition. For the decomposition of DCP, the interaction between dual computers CP_0 and CP_1 and fault hypotheses $FHYP$ is specified by the following

ARCHITECTURAL DECOMPOSITION $\mathcal{D}[DCP]$:

IMPORT: $\mathcal{I}(CP_0), \mathcal{I}(CP_1), \mathcal{I}[FHYP]$
DECOMPOSITION:

$$DCP = D_{DCP}(CP_0, CP_1, FHYP)$$
$$D_{DCP}(CP_0, CP_1, FHYP) =_{df}$$
$$((CP_0 \underset{\mathit{off}_0}{\nabla} RUN(\{\| a, c_0 \|\})) \underset{\{\| \mathit{off}_0, c_0, c_1, a \|\}}{\|} CP_1) \underset{\{\mathit{off}_0\}}{\|} FHYP$$

As indicated above, the two computers exchange control messages on channels c_0, c_1. The event off_0 denotes failure of CP_0 which then performs a transition into the idle mode $RUN(\{\| a, c_0 \|\})$ indicating that CP_0 will stop to produce outputs on b or c_1, but also won't block any a- or c_0-communications after a crash. Failure event off_0 it is also detected by CP_1 which initiates the recovery activities in the standby computer. This will become apparent in the next decomposition step.

Requirements Specifications. We start with the specification of $FHYP$. Our simplified example covers the situation where computer CP_0 is initialised to act as the master computer. The master may or may not fail, but in order to keep the example small, we do not consider a crash of the standby computer CP_1 or the re-integration of repaired components and consecutive failure situations (this is described in [18]). These assumptions are modelled explicitly by the following specification.

REQUIREMENTS SPECIFICATION $\mathcal{R}[FHYP]$:

IMPORT: $\mathcal{I}[FHYP]$
EXPLICIT REQUIREMENT: $FHYP = \mathit{off}_0 \rightarrow STOP \sqcap STOP$

Process $FHYP$ admits at most one crash of CP_0. Since the failure event off_0 is synchronised between $FHYP$ and the "real" components CP_0, CP_1, the auxiliary process restricts the set of all executions possible for $(CP_0 \underset{\{\mid \mathit{off}_0, c_0, c_1, a \mid\}}{\|} CP_1)$ to those where at most one crash happens. Observe that it is not intended to *implement* $\mathcal{R}[FHYP]$. The specification just models our assumptions about the faults that may occur, and we prove that the dual computer system acts according to its specification *as long as these assumptions are valid*. For these reasons, $\mathcal{R}[FHYP]$ is depicted by a dashed-line box in Figure 2.

To motivate the requirements specifications for CP_0 and CP_1 below, it is important to look ahead at the strategy to be applied in the proof that the dual computers implement the higher-level requirements $\mathcal{R}[DCP]$: Our plan is to separate the full verification obligation $\mathcal{V}[DCP]$ arising from the present decomposition step into two parts, where the first can be verified by simple compositional reasoning, while the second, more complex, part can be verified by model checking in a completely automatic way. Now "classical" model checking as supported by the FDR tool requires finite-state processes, finite (preferably small) data ranges and does not admit generic data types $DATA$ or transformations f. Therefore the explicit requirements specifications to be analysed by the model checker must refer to *finite-state/finite-value abstractions* of CP_i. For our purposes, a simple abstraction type is already powerful enough (more complex abstraction techniques have been described in [6]): It can be expressed by renaming of events, and the renaming is performed in such a way that all references to generic data types and functions vanish in the abstraction process. Formally, our abstraction mapping is given by

ABSTRACTION DEFINITION $\mathcal{A}$:

ABSTRACTION:

$$A(X) =_{df} X[\![\, a.x.bit \leftarrow a.bit, b.x.bit \leftarrow b.bit, a_{i2}.x \leftarrow a_{i2}, b_{i2}.x \leftarrow b_{i2} \\ \mid x : DATA;\ bit : BIT;\ i : \{0,1\}]\!]$$

In this definition, $P[\![c \leftarrow d]\!]$ denotes process P with all occurrences of c replaced by d. With these preliminary considerations, the specification of master computer CP_0 is given by

REQUIREMENTS SPECIFICATION $\mathcal{R}[CP_0]$:

IMPORT: $\mathcal{I}[CP_0]$
IMPLICIT REQUIREMENT: $CP_0 \setminus (\Sigma - I(CP_0))$ **sat** $S'_{DCP}(s, R)$
IMPLICIT SPECIFICATION:

$$
\begin{aligned}
S'_{DCP}(s) \equiv_{df} & \ (\exists\, y : DATA;\ d : BIT \bullet last(s) = b.y.d) \\
& \Rightarrow y = f(data(last(s \restriction a)))
\end{aligned}
$$

EXPLICIT REQUIREMENT: $X_0 \sqsubseteq_{FD} A(CP_0 \setminus (\Sigma - I(CP_0))$
EXPLICIT SPECIFICATION: $X_0 = a?d \rightarrow b!d \rightarrow c_1 \rightarrow X_0$

The implicit part of $\mathcal{R}[CP_0]$ is generic in $DATA$ and f and states everything we need to ensure about the data transformation behaviour of CP_0: The computer always outputs on channel b the most recent data input received on a, transformed by function f. The implicit specification is denoted by $S'_{DCP}(s)$, because together with another specification $S''_{DCP}(s)$ to be defined below, it implies the implicit specification $S_{DCP}(s, R)$ defined for DCP in $\mathcal{R}[DCP]$. The explicit part of CP_0's requirements specifications refers to input/output sequencing and to the bit-values passed along channels a and b: As a refinement of X, CP_0 will always service its communication channels in the order $a, b, c_1, a, \ldots$. The most recent bit-value d received on a will be returned to the environment on b.

For the specification of CP_0, the requirements related to data transformations are the same, since the computer must be capable of performing exactly the same service as CP_0. However, the explicit specification is more complex, since it also models the reaction on failure event off_0 indicating the crash of master computer CP_0 (in [18], fully symmetric specifications for CP_i are given, dealing with the failure of both computers). Requirements $\mathcal{R}[CP_1]$ are given by

REQUIREMENTS SPECIFICATION $\mathcal{R}[CP_1]$:

IMPORT: $\mathcal{I}[CP_1]$
IMPLICIT REQUIREMENT: $CP_1 \setminus (\Sigma - I(CP_1))$ **sat** $S'_{DCP}(s, R)$
IMPLICIT SPECIFICATION:

$$\begin{aligned} S'_{DCP}(s) \equiv_{df} & \ (\exists\, y : DATA;\ d : BIT \bullet last(s) = b.y.d) \\ & \Rightarrow y = f(data(last(s \upharpoonright a))) \end{aligned}$$

EXPLICIT REQUIREMENT: $X_1 \sqsubseteq_{FD} A(CP_1 \setminus (\Sigma - I(CP_1))$
EXPLICIT SPECIFICATION:

$$\begin{aligned} X_1 &= a?d \to (c_1 \to X_1 \,\Box\, \mathit{off}_0 \to b!d \to c_0 \to X_1') \\ &\quad \Box\, \mathit{off}_0 \to X_1' \\ X_1' &= a?d \to b!d \to c_0 \to X_1' \end{aligned}$$

The explicit specification states that CP_1 stores the most recent bit received on a until an acknowledge from CP_0 is received on c_1. If a crash occurs before this acknowledge has been received, the associated b-output is repeated. After a crash, CP_1 services channels a, b, c_1 in analogy to CP_0.

8.2 Verification of the DCP-Decomposition: Compositional Reasoning and Model Checking

The invent step for DCP performed in the previous section give rise to the following verification obligation:

VERIFICATION OBLIGATION $\mathcal{V}[DCP]$:

IMPORT: $\mathcal{R}[DCP], \mathcal{R}[CP_0], \mathcal{R}[CP_1], \mathcal{R}[FHYP], \mathcal{D}[DCP]$

HIGHER-LEVEL REQUIREMENTS:
$\mathcal{R}[DCP]$
IMPLIED BY:
lower-level requirements:
$\mathcal{R}[CP_0], \mathcal{R}[CP_1], \mathcal{R}[FHYP]$
architectural decomposition:
$\mathcal{D}[DCP]$

Strategy Selection. As indicated above, our goal is to decompose requirement $\mathcal{R}[DCP]$ into two parts in order to separate specifications referring to generic data types and functions from those referring only to finite data ranges and sequencing of events. The two specifications are defined by

$$S'_{DCP}(s) \equiv_{df} (\exists\, y : DATA;\ d : BIT \bullet last(s) = b.y.d)$$
$$\Rightarrow y = f(data(last(s \upharpoonright a)))$$

$$
\begin{aligned}
&S''_{DCP}(s, R) \equiv_{df} \\
&\quad NO\_BLOCK[(a), (b)](s, R) \\
&\quad \wedge\ ch^*(s) \in \mathbf{pref}\langle \mathbf{a}, [\mathbf{b} \mid \mathbf{b}, \mathbf{b}]\rangle^* \\
&\quad \wedge\ ((\exists\, d : BIT \bullet (ch \times bit)(last(s)) = b.d)) \Rightarrow d = bit(last(s \upharpoonright a)))
\end{aligned}
$$

where $S'_{DCP}(s)$ has already been introduced above. Specification $S''_{DCP}(s, R)$ states that

1. $DCP \setminus (\Sigma - I(DCP))$ will never deadlock on its interface $\{\!|\ a, b\ |\!\}$,
2. $DCP \setminus (\Sigma - I(DCP))$ always accesses its interface channels in the order defined by the regular expression $\langle a, [b \mid b, b]\rangle^*$, that is, each a-event will be followed by one or two b-events,
3. $DCP \setminus (\Sigma - I(DCP))$ will always output on b the last bit value received on channel a.

Our approach is justified by the following lemma:

Lemma 8.1. *If DCP **sat** $S'_{DCP}(s, R) \wedge S''_{DCP}(s, R)$ holds, then DCP **sat** $S_{DCP}(s, R)$ follows.*

Proof. The NO_BLOCK and $LEFT\_GUARD$ conjuncts in $S_{DCP}(s, R)$ are directly implied by $S''_{DCP}(s, R)$. The two specification $S_{DCP}(s)'$ and $S''_{DCP}(s, R)$ together imply that every trace of DCP performed at its interface $\{\!|\ a, b\ |\!\}$ is a prefix of

$$\langle a.(x_1, d_1), [b.(f(x_1), d_1) \mid b.(f(x_1), d_1), b.(f(x_1), d_1)],$$
$$a.(x_2, d_2), [b.(f(x_2), d_2) \mid b.(f(x_2), d_2), b.(f(x_2), d_2)], \ldots\rangle$$

with suitable data values x_i and bit values d_i. This implies $data^* \circ \rho(s \upharpoonright b) \leq^1 f^* \circ data^* \circ \rho(s \upharpoonright a)$, so that the $COPY$ predicate in $S_{DCP}(s, R)$ is also induced by the new specifications. □

Since $S'_{DCP}(s)$ is a trace assertion, it should be easy to verify this property using the compositional proof theory of CSP interpreted in the trace model. Specification $S''_{DCP}(s)$ does not refer to $DATA$ and f. As a consequence, our definition $\mathcal{A}$ specifies a *valid abstraction* for specification $S''_{DCP}(s)$ in the following sense[4]:

For all processes P, $A(P)$ **sat** $S''_{DCP}(s)$ implies P **sat** $S''_{DCP}(s)$.

Since satisfaction relations are preserved under refinement, our verification strategy can now be summarised as follows:

[4] Observe that since it does not refer to values $y : DATA$, specification $S''_{DCP}(s)$ is well-defined for both traces over the original channel events $a.x.d, b.x.d$ and the abstracted events $a.d, b.d$. Otherwise it would be necessary to define a "lifted" specification $A^*(S''_{DCP}(s))$ defined on the abstracted trace space.

1. Use compositional reasoning to prove that the implicit specification parts of $\mathcal{R}[CP_i]$ imply that $D_{DCP}(CP_0, CP_1, FHYP)$ satisfies $S'_{DCP}(s)$.
2. Find a simple explicit process Y obviously satisfying $S''_{DCP}(s, R)$ and prove by model checking that $Y \sqsubseteq A(D_{DCP}(X_0, X_1, FHYP))$ with X_i defined as in $\mathcal{R}[CP_i]$.
3. Conclude from step 2 that the decomposition $D_{DCP}(CP_0, CP_1, FHYP)$ satisfies $S''_{DCP}(s, R)$.

Verification by Compositional Reasoning. Since $\mathcal{R}[CP_i]$ state that computers CP_i shall both satisfy $S_{DCP}(s)'$, the first part of our verification obligations is expressed by the following lemma which can be proved using our HOL encoding:

Lemma 8.2. *With the definitions given in* $\mathcal{R}[DCP]$, $\mathcal{R}[CP_0]$, $\mathcal{R}[CP_1]$,$\mathcal{R}[FHYP]$ *and* $\mathcal{D}[DCP]$, $CP_0 \setminus (\Sigma - I(CP_0))$ **sat** $S_{DCP}(s)'$ *and* $CP_1 \setminus (\Sigma - I(CP_1))$ **sat** $S_{DCP}(s)'$ *implies* $DCP \setminus (\Sigma - I(DCP))$ **sat** $S'_{DCP}(s)$

Proof. According to the decomposition $DCP = D_{DCP}(CP_0, CP_1, FHYP)$ defined in $\mathcal{D}[DCP]$, we have to show that

$$D_{DCP}(CP_0, CP_1, FHYP) \setminus (\Sigma - I(DCP)) \ \mathbf{sat} \ S'_{DCP}(s)$$

holds. Specification $S_{DCP}(s)'$ refers only to channels a and b which are not hidden in any of the three processes. Therefore the proof is performed by multiple application of the compositional law

$$\begin{aligned} &P \ \mathbf{sat} \ S_P(h) \text{ and } Q \ \mathbf{sat} \ S_Q(h) \\ &\qquad\qquad\text{implies } (P \parallel Q) \ \mathbf{sat} \ (S_P(h \restriction \alpha(P)) \wedge S_Q(h \restriction \alpha(Q))) \end{aligned}$$

which holds in the trace model of CSP: CP_0 and CP_1 both satisfy $S'_{DCP}(s)$ and synchronise over a, as a consequence $S'_{DCP}(s)$ is also satisfied by their parallel composition. Moreover, $FHYP$ and the interrupt $\underset{\mathit{off}_0}{\nabla}$ do not produce any a- or b-events. This proves the lemma. □

Verification by Model Checking. Next, we turn to the second specification, and it will be shown that its validity is implied by the explicit specification parts in $\mathcal{R}[CP_0]$ and $\mathcal{R}[CP_1]$. This is stated in the following lemma:

Lemma 8.3. *With the definitions given in* $\mathcal{R}[DCP]$, $\mathcal{R}[CP_0]$, $\mathcal{R}[CP_1]$,$\mathcal{R}[FHYP]$ *and* $\mathcal{D}[DCP]$, $X_0 \sqsubseteq_{FD} A(CP_0 \setminus (\Sigma - I(CP_0))$ *and* $X_1 \sqsubseteq_{FD} A(CP_1 \setminus (\Sigma - I(CP_1))$ *implies* $DCP \setminus (\Sigma - I(DCP))$ **sat** $S''_{DCP}(s, R)$

Proof. In analogy to the previous lemma, we have to show that

$$D_{DCP}(CP_0, CP_1, FHYP) \setminus (\Sigma - I(DCP)) \ \mathbf{sat} \ S''_{DCP}(s, R)$$

holds. Following the strategy sketched above, we start by inventing an auxiliary process Y which obviously satisfies $S''_{DCP}(s, R)$. Such a process is given by

$$Y = a?bit \rightarrow (b!bit \rightarrow Y \sqcap b!bit \rightarrow b!bit \rightarrow Y)$$

(This could also be checked using the transformation of Y into the corresponding nondeterministic sequential program to be verified by the PAMELA/PVS tool.)

Failures-Divergence refinement preserves satisfaction relations and it has been noted above that $A(P)$ **sat** $S''_{DCP}(s, R)$ implies P **sat** $S''_{DCP}(s, R)$ for all processes P. As a consequence, it suffices to verify

$$Y \sqsubseteq_{FD} A(D_{DCP}(CP_0, CP_1, FHYP) \setminus (\Sigma - I(DCP)))$$

in order to prove the lemma. Next, we observe that

$$\begin{aligned} &A(D_{DCP}(CP_0, CP_1, FHYP) \setminus (\Sigma - I(DCP))) \\ &\qquad = D_{DCP}(A(CP_0), A(CP_1), FHYP) \setminus (\Sigma - I(DCP)) \end{aligned}$$

where channels a and b occurring in the terms on the right-hand side of the equation denote the abstracted versions without $DATA$-components x.

Since refinement is compositional, the refinement relations $X_i \sqsubseteq_{FD} A(CP_i \setminus (\Sigma - I(CP_i))$ asserted in $\mathcal{R}[CP_i]$ imply

$$\begin{aligned} &D_{DCP}(X_0, X_1, FHYP) \setminus (\Sigma - I(DCP)) \sqsubseteq_{FD} \\ &\quad D_{DCP}(A(CP_0 \setminus (\Sigma - I(CP_0)), A(CP_1 \setminus (\Sigma - I(CP_1)), A(FHYP)) \\ &\qquad \setminus (\Sigma - I(DCP)) \end{aligned}$$

The right-hand side of this refinement relation is identical to $D_{DCP}(A(CP_0), A(CP_1), FHYP) \setminus (\Sigma - I(DCP))$, because $I(DCP)$ is a subset of both $I(CP_0)$ and $I(CP_1)$. Since $\sqsubseteq_{FD}$ is transitive, the desired refinement relation $Y \sqsubseteq_{FD} D_{DCP}(A(CP_0), A(CP_1), FHYP) \setminus (\Sigma - I(DCP))$ would follow if we could prove

$$Y \sqsubseteq_{FD} D_{DCP}(X_0, X_1, FHYP) \setminus (\Sigma - I(DCP))$$

Now this refinement relation does not contain any free process variables or generic data types or functions. As a consequence, it can be verified by model checking using the FDR tool. This check has been performed and does not show any errors, which proves the lemma. $\square$

Together, Lemma 8.1, 8.2 and 8.3 discharge verification obligation $\mathcal{V}[DCP]$.

9 Invent&Verify Steps for CP_0

9.1 Decomposition of CP_0 Into Processes NET_0 and APP_0

As indicated in Figure 1, each component of the dual computer system is devided into two layers: A network process NET_i performs the communication with the environment of DCP and exchanges synchronisation events. An application process APP_i performs the computations $y := f(x)$ on the active computer. For CP_0, this decomposition and its verification will be performed in the present section.

Interface Definitions. The interface of the networking component NET_0 is defined by

INTERFACE DEFINITION $\mathcal{I}[NET_0]$:

DEFINITION:	$I(NET_0) =_{df} \{\| a, b, c_0, c_1, a_{02}, b_{02} \|\}$
INPUTS:	a, c_0, b_{02}
OUTPUTS:	b, c_1, a_{02}

The application process APP_0 has interface

INTERFACE DEFINITION $\mathcal{I}[APP_0]$:

DEFINITION:	$I(APP_0) =_{df} \{\| a_{02}, b_{02} \|\}$
INPUTS:	a_{02}
OUTPUTS:	b_{02}

Architectural Decomposition of CP_0. The two layers operate in parallel and use channels a_{02}, b_{02} for internal communication. This gives rise to the following architectural decomposition:

ARCHITECTURAL DECOMPOSITION $\mathcal{D}[CP_0]$:

IMPORT: $\mathcal{I}(NET_0), \mathcal{I}(APP_0)$
DECOMPOSITION:

$$CP_0 = D_{CP_0}(NET_0, APP_0)$$

$$D_{CP_0}(NET_0, APP_0) =_{df} (NET_0 \underset{\{\| a_{02}, b_{02} \|\}}{\|} APP_0)$$

Explicit Specifications. Both networking and application layers can be explicitly specified as implementable CSP processes.

IMPLEMENTABLE PROCESS SPECIFICATION $\mathcal{P}[NET_0]$:

IMPORT: $\mathcal{I}[NET_0]$
EXPLICIT SPECIFICATION:

$$\begin{aligned} NET_0 = \ & a?(x, bit) \to a_{02}!x \to b_{02}?y \\ & \to b!(y, bit) \to c_1 \to NET_0 \end{aligned}$$

Process NET_0 accepts input data and alternating bits on a. After that, the data part is offered to the application layer on channel a_{02}. If the application layer is active (that is, the local computer acts as master) it will accept the data, and NET_0 gets the transformed value y back on channel b_{02}. The value y is again associated with the bit of the corresponding input message, and the resulting tuple is sent back to the client via channel b. After that, successful completion of the transaction is signalled to the standby computer via channel c_1. Since in the simplified example CP_0 never acts as standby, NET_0 never accepts control signals c_0 from NET_1, as long as CP_0 is active.

The explicit requirements for APP_0 just describe the desired data transformation behaviour:

IMPLEMENTABLE PROCESS SPECIFICATION $\mathcal{P}[APP_0]$:

IMPORT: $\mathcal{I}[APP_0]$

EXPLICIT SPECIFICATION: $APP_0 = a_{02}?x \to b_{02}!f(x) \to APP_0$

9.2 Verification of the CP_0-Decomposition: Compositional Reasoning and Model Checking

The verification obligation to be discharged for the present development step is

VERIFICATION OBLIGATION $\mathcal{V}[CP_0]$:

IMPORT: $\mathcal{R}[CP_0], \mathcal{P}[NET_0], \mathcal{P}[APP_0], \mathcal{D}[CP_0]$

HIGHER-LEVEL REQUIREMENTS:

$\mathcal{R}[CP_0]$

IMPLIED BY:

lower-level requirements:

$\mathcal{P}[NET_0], \mathcal{P}[APP_0]$

architectural decomposition:

$\mathcal{D}[CP_0]$

Strategy Selection. $\mathcal{R}[CP_0]$ contains both an implicit and an explicit requirement. The implicit one will be discharged by compositional reasoning about NET_0, APP_0 and their parallel composition. The explicit requirement can be verified again by model checking.

Verification by Compositional Reasoning. The proof obligation to be discharged for the implicit requirement in $\mathcal{V}[CP_0]$ is given by the following lemma:

Lemma 9.1. *With the definitions given in $\mathcal{R}[CP_0]$, $\mathcal{P}[NET_0]$, $\mathcal{P}[APP_0]$, and $\mathcal{D}[CP_0]$, the explicit specifications for NET_0 and APP_0 imply*

$$CP_0 \setminus (\Sigma - I(CP_0)) \ \mathbf{sat} \ S'_{DCP}(s)$$

Proof. According to the architectural decomposition $\mathcal{D}[CP_0]$, we have to show that

$$D_{CP_0}(NET_0, APP_0) \setminus (\Sigma - I(CP_0)) \ \mathbf{sat} \ S'_{DCP}(s)$$

The definition of $S'_{DCP}(s)$ is independent on other channels but a and b, so the proof obligation can be re-formulated as

$$(NET_0 \underset{\{\mid a_{02}, b_{02} \mid\}}{\|} APP_0) \ \mathbf{sat} \ S'_{DCP}(s)$$

with explicit specifications for NET_0 and APP_0 according to $\mathcal{P}[NET_0]$ and $\mathcal{P}[APP_0]$. As in Lemma 8.2, we will make use of the compositional law for the parallel operator in the trace model of CSP. First we observe that the auxiliary assertion

$$NET_0 \ \mathbf{sat} \ S_{NET_0}(s)$$

$$\begin{aligned} & S_{NET_0}(s) \equiv_{df} \\ & \quad ((\exists\, x : DATA \bullet last(s) = a_{02}.x) \Rightarrow x = data(last(s \upharpoonright a))) \\ & \quad \wedge \\ & \quad ((\exists\, y : DATA;\ d : BIT \bullet last(s) = b.(y, d)) \Rightarrow y = data(last(s \upharpoonright b_{02}))) \end{aligned}$$

is obviously satisfied. This follows directly from the process definition of NET_0 in $\mathcal{P}[NET_0]$, but could also be mechanically proven using PAMELA/PVS. Next we note that

$$APP_0 \ \mathbf{sat} \ S_{APP_0}(s)$$

$$\begin{aligned} & S_{APP_0}(s) \equiv_{df} \\ & \quad (\exists\, y : DATA \bullet last(s) = b_{02}.y) \Rightarrow y = f(data(last(s \upharpoonright a_{02}))) \end{aligned}$$

is also satisfied as follows directly from $\mathcal{P}[APP_0]$. Application of the $\|$-law results in

$$(NET_0 \underset{\{\mid a_{02}, b_{02} \mid\}}{\|} APP_0) \ \mathbf{sat} \ S_{NET_0}(s) \wedge S_{APP_0}(s \upharpoonright \{\mid a_{02}, b_{02} \mid\})$$

and this directly implies satisfaction of $S'_{DCP}(s)$. □

Verification by Model Checking. To verify the explicit requirement stated in $\mathcal{R}[CP_0]$, we will use model checking. The proof obligation is given in the following lemma:

Lemma 9.2. *With the definitions given in $\mathcal{R}[CP_0]$, $\mathcal{P}[NET_0]$, $\mathcal{P}[APP_0]$, and $\mathcal{D}[CP_0]$, the explicit specifications for NET_0 and APP_0 imply*

$$X_0 \sqsubseteq_{FD} A(CP_0 \setminus (\Sigma - I(CP_0))$$

Proof. Evaluating the architectural decomposition $\mathcal{D}[CP_0]$ and the definitions of A, NET_0 and APP_0 results in the revised proof obligation

$$X_0 \sqsubseteq_{FD} (A(NET_0) \underset{\{\!|\, a_{02}, b_{02} \,|\!\}}{\|} A(APP_0)) \setminus \{\!|\, a_{02}, b_{02} \,|\!\}$$

with

$$A(NET_0) = a?bit \to a_{02} \to b_{02} \to b!bit \to c_1 \to A(NET_0)$$

and

$$A(APP_0) = a_{02} \to b_{02} \to c_1 \to A(APP_0)$$

This refinement relation has been mechanically checked using the FDR tool which showed no errors. This completes the proof of the lemma □

Together, lemmas 9.1 and 9.2 discharge verification obligation $\mathcal{V}[CP_0]$.

10 Invent&Verify Steps for CP_1

10.1 Decomposition of CP_1 Into Processes NET_1 and APP_1

In analogy to CP_0, computer CP_1 is structured into networking process NET_1 and application APP_1.

Interface Definitions. The interfaces of NET_1 and APP_1 are defined in analogy to those of NET_0 and APP_0.

Architectural Decomposition of CP_1. As for CP_0, the two layers operate in parallel and use channels a_{12}, b_{12} for internal communication. However, the application layer of CP_1 will only become active as a reaction to the crash event off_0, indicating that CP_0 is no longer able to process jobs.

ARCHITECTURAL DECOMPOSITION $\mathcal{D}[CP_1]$:

IMPORT: $\mathcal{I}(NET_1), \mathcal{I}(APP_1)$
DECOMPOSITION:

$$CP_1 = D_{CP_1}(NET_1, APP_1)$$

$$D_{CP_1}(NET_1, APP_1) =_{df} (NET_1 \underset{\{\!|\, a_{12}, b_{12} \,|\!\}}{\|} \mathit{off}_0 \to APP_1)$$

Explicit Specifications. The explicit specification $\mathcal{P}[APP_1]$ is analogous to $\mathcal{P}[APP_0]$ defined above. The networking component of CP_1 has to react on acknowledgement signals from CP_0:

IMPLEMENTABLE PROCESS SPECIFICATION $\mathcal{R}[NET_1]$:

IMPORT: $\mathcal{I}[NET_1]$
EXPLICIT SPECIFICATION:

$$\begin{aligned} NET_1 = a?(x, bit) \rightarrow (&c_1 \rightarrow NET_1 \\ &\Box \\ &a_{12}!x \rightarrow b_{12}?y \rightarrow b!(y, bit) \rightarrow c_1 \rightarrow NET_1) \end{aligned}$$

Observe that NET_1 can only enter the $\Box$-branch $a_{12}!x \rightarrow \ldots$ if the application APP_1 is active, that is, after the failure event off_0 has occurred.

We skip the definition and verification of obligation $\mathcal{V}[CP_1]$, since it is completely analogous to that of $\mathcal{V}[CP_0]$. This completes the development and verification of the dual computer system according to the invent&verify paradigm.

11 Conclusion

Presenting the development of a dual computer system, we have illustrated an approach for the tool-based verification of fault-tolerant systems using combinations of specification formalisms, proof strategies and associated verification tools. Our main objective was to motivate that this combination can be effectively applied in complex "real-world" developments. This claim is supported by experiences with the verification of a fault-tolerant system developed by Daimler-Benz Aerospace for the International Space Station ISS [6]. It should be noted that the verification approach presented in this article is complemented by automated hardware-in-the-loop tests performing real-time tests for the completed hardware and software system [25] using the VVT-RT test tool.

To further increase the efficiency of the verification approach described, the tool chain is presently extended in the following directions:

- automatic transformation of CSP specifications to abstracted versions admissible for a specific verification purpose,
- systematic development of data structures and sequential algorithms using Z,
- specification and verification of hybrid requirements referring to time-continuous values.

At present, the development and verification strategies described as well as the tools supporting them are integrated in the UniForM Workbench [17]. The navigation between development and verification objects within the workbench is implemented by the DaVinci tool [13] providing a user interface displaying development graphs. To perform a specific development or verification step, the interactive graph representation allows to activate tools encapsulated in the workbench, such as HOL, FDR, PAMELA/PVS and the VVT-RT test tool.

References

1. K. R. Apt, L. Bougé and Ph. Clermont: Two Normal Form Theorems for CSP Programs. *Information Processing Letters* (1987) 26:165-171.
2. 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.
3. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, Berlin Heidelberg New York (1991).
4. B. Buth: *Operation Refinement Proofs for VDM-like Specifications*, Dissertation; auch Bericht Nr. 9501 Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel 1995.
5. B. Buth: PAMELA + PVS. In Michael Johnson (Ed.): Algebraic Methodology and Software Technology. Proceedings of the AMAST'97, Sidney, Australia, December 1997, Springer LNCS 1349 (1997), 560-562.
6. Bettina Buth, Michel Kouvaras, Jan Peleska, Hui Shi: Deadlock Analysis for a Fault-Tolerant System In Michael Johnson (Ed.): Algebraic Methodology and Software Technology. Proceedings of the AMAST'97, Sidney, Australia, December 1997, Springer LNCS 1349 (1997), 60-75.
7. Bettina Buth, Rachel Cardell-Oliver and Jan Peleska: Combining Tools for the Verification of Fault-Tolerant Systems. Technical Report, Bremen Institute of Safe Systems BISS (1998).
8. Camilleri, Albert.J: A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics, In G. Birtwistle (ed), *Workshops in Computing, Banff, Canada, Sept 1990*, Springer-Verlag, 1991, pp123–150.
9. Cardell-Oliver, Rachel: Using Higher Order Logic for Modelling Real-Time Protocols, In: *TAPSOFT 91, Vol.2*, Lecture Notes in Computer Science No. 494, Springer Verlag, 1991, pp259–282.
10. Cardell-Oliver, R.M: *The Formal Verification of Hard Real-Time Systems*, PhD Thesis, University of Cambridge, (1992).
11. Davies, J: *Specification and Proof in Real-Time CSP.* Cambridge University Press (1993).
12. Formal Systems: *Failures-Divergence Refinement.* FDR2 User Manual Formal Systems (Europe) Ltd (1997).
13. M. Frhlich, M. Werner: Demonstration of the interactive Graph Visualization System daVinci. In: R. Tamassia, I. Tollis: *Proceedings of DIMACS Workshop on Graph Drawing '94, Princeton (USA) 1994*; Lecture Notes in Computer Science No. 894; Springer Verlag; January (1995). See also `http://www.informatik.uni-bremen.de/~davinci`
14. Gordon, M.J.C., Melham, T.F: *Introduction to the HOL System*, Cambridge University Press (1994)
15. 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.

16. C.A.R. Hoare: *Communicating Sequential Processes*; Prentice Hall International, (1985).
17. Krieg-Brckner, B., Peleska, J., Olderog, E.-R., Balzer, D., Baer, A.: UniForM, Universal Formal Methods Workbench. In Grote, U. (ed.): Statusseminar Softwaretechnologie. BMBF (1997). See also `http://www.informatik.uni-bremen.de/~uniform`
18. J. Peleska: *Formal Methods and the Development of Dependable Systems*; *Habilitationsschrift*, Technical Report 9612, Institut für Informatik, Christian-Albrechts-Universität, Kiel (1996).
19. J. Peleska and M. Siegel: Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal* (1997)19:53-77.
20. A. P. Ravn: *Design of Embedded Real-Time Computing Systems*, Dänische Doktorarbeit, 1995
21. A. W. Roscoe: Model-Checking CSP. In *A Classical Mind, Essays in Honour of CAR Hoare*. Prentice-Hall International, Englewood Cliffs NJ (1994).
22. A. W. Roscoe: Developing and Verifying Protocols in CSP In *Protocol Verification Workshop, Mierlo, The Netherlands, March 1993* (1993).
23. A. W. Roscoe: *The Theory and Practice of Concurrency.* Prentice Hall International, (1997).
24. John Rushby: *A tutorial on specification and verification using PVS*, In: Tutorial Material for FME '93: Industrial-Strength Formal Methods. Proceedings of the First International Symposium of Formal Methods Europe, Odense, Denmark, 1993.
25. Gerd Urban, Hans-Joachim Kolinowitz and Jan Peleska: A Survivable Avionics System for Space Applications. Submitted to FTCS-28, 28th Annual Symposium on Fault-Tolerant Computing, Munich, Germany, 1998.