Applied Formal Methods - From CSP to Executable Hybrid Specifications

Jan Peleska

University of Bremen,
P.O. Box 330 440
28334 Bremen, Germany
and
Verified Systems International GmbH
jp@verified.de

Abstract. Since 1985, CSP has been applied by the author, his research team at Bremen University and verification engineers at Verified Systems International to a variety of "real-world" projects. These include the verification of high-availability database servers, of fault-tolerant computers now operable in the International Space Station, hardware-in-the-loop tests for the novel Airbus A380 aircraft controller family and conformance tests for the European Train Control System. Illustrated by examples from these projects, we highlight important aspects of the CSP language design, its semantics and tool support, and describe the impact of these features on the quality and efficiency of verification and testing. New requirements with regard to the test of hybrid control systems, the demand for executable formal specifications, as well as the ongoing discussion about the practical applicability of formal methods have led to the development of new specification formalisms. We sketch some key decisions in the formalism design and indicate how some of the fundamental properties of CSP have been adopted, while others have been deliberately discarded in these new developments.

1 Introduction

Motivation. The objectives of this contribution are twofold. First, we wish to illustrate the usability of *Communicating Sequential Processes CSP* for specification, verification and testing in an industrial "real-world" context. Second, we are convinced that further research work on CSP and similar formalisms will benefit from the challenges which are posed by problems occurring in daily industrial verification practice.

Overview. In Section 2, an overview of industrial verification projects managed by the author is given. In each of these projects, CSP served as the underlying formalism for specification, verification and testing. We sketch how existing methods and theories contributed to the solution of each problem, and how the "feed-back loop" between research and industrial projects was closed by practical problems leading to novel research challenges. In the two sections to follow,

more recent related research activities are described: Section 3 outlines recent results and ongoing research work in the field of automated testdata generation from Timed CSP (TCSP) specifications. In Section 4 we introduce a framework for generating run-time environments for real-time execution of specifications written in "high-level" formalisms, such as TCSP, Statecharts, further diagram types of the Unified Modeling Language and Hybrid systems extensions thereof. The latter allow to describe both time-discrete changes and analog evolutions of physical observables. The framework, which is currently used for specification-based testing against various formalisms – TCSP is one of them – has been developed for the purpose of model-based development, test data generation and on-the-fly checking of system behaviour. Section 5 contains the conclusion.

Further Reading. Some basic knowledge about CSP in its untimed and timed variants is assumed in this article. For a detailed introduction readers are referred to [Hoa85,Ros98,Sch00]. References to further research results which are of interest within the scope of this paper are given in the sections below.

2 Practice Stimulates Theory – Applied CSP and Related Research Activities

2.1 Specification and Verification of Fault-Tolerant Systems

The Dual Computer System DCP. In 1985 the author and his team at Philips started the design and development of a fault-tolerant dual computer system *DCP* for a high-availability data base server. A design novelty at that time consisted in using a more symmetric concept than the usual master-standby technique: Both computers $CP_i, i = 0, 1$ of the *DCP* were active during normal operation without being strictly synchronised on instruction level. They both executed read-write transactions on their local data bases, but read-only transactions were executed by just one of them, while the other only stored the inputs from the client until the associated transaction had been completed. This strategy could be exploited for higher performance in comparison to a server consisting of only one computer. To minimise the number of synchronisation messages to be exchanged and processed between them, the two computers only synchronised their serialisation for conflicting read-write transactions[1], so that a consistent database state was maintained. As long as both computers were active only one of them returned transaction results back to the client. If CP_i failed, computer $CP_{(1-i)}$ only had to redo the open read-only transactions performed by CP_i – this was possible because $CP_{(1-i)}$ still kept the associated input data – and to transmit the results of all transactions which had not yet been sent to the client when failure occurred. This strategy avoided loss of transactions or messages from server to client, but it could lead to duplicate messages.

[1] Two transactions $T_i, i = 0, 1$ are called *conflicting* if the write set of T_i has a non-empty intersection with the union of $T_{(1-i)}$'s read and write sets.

These could be filtered by implementing an alternating bit protocol for client-server communication. After delivering the results of all open transactions to the clients, the remaining computer $CP_{(1-i)}$ could provide all database services since its local database was up-to-date. The only degradation visible to clients consisted in slower response time, since now all read-only transactions had to be performed on a single computer. Figure 1 sketches the *DCP* architecture; the fault-tolerance mechanisms are encapsulated in a separate layer denoted by $NET_i, i = 0, 1$. Specification details are available in [Pel97, pp. 59].

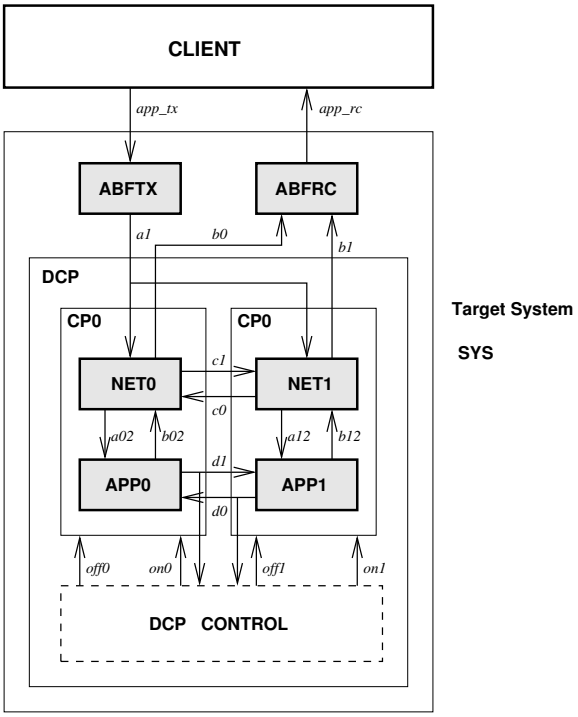

Fig. 1. Architecture of the dual computer system *DCP*.

Verification Strategy for the DCP. The complexity of the fault-tolerance services which were needed to implement this type of behaviour suggested that a rigorous verification approach should complement the conventional testing activities planned for the *DCP*. At that time, Tony Hoare's book [Hoa85] on CSP became available, and later a joint publication with He Jifeng [JH87] described an elegant verification technique for fault-tolerant systems: Using purely algebraic reasoning within the CSP process algebra, the authors showed that the fault-tolerant implementation process and the associated requirements specifi-

cation process both satisfied the same set of mutually recursive equations. Now equivalence between the two followed simply from fixed-point theory.

During initial formal verification attempts, however, it was realised that – rather than applying a single verification technique for all tasks – it would be more efficient to use a combination of specification and verification "styles", so that for each step within the verification suite the most promising technique could be selected:

1. The top-level requirements were formulated in implicit specification style, as a proof obligation on traces and refusals: SYS **sat** $S(tr, R)$.
2. Following a top-down decomposition of the system design sketched in Figure 1, each component was first associated with local proof obligations about its interface behaviour.
3. Using compositional reasoning for the parallel and hiding operators (see, for example, [Sch00, pp. 197]) it was shown in each decomposition step that the required behaviour of sub-components would imply the proof obligation specified for the higher-level component.
4. When the stepwise decomposition reached the level of sub-components to be implemented as sequential processes P_i, these processes were not only associated with their implicit specifications P_i **sat** $S_i(tr, R)$, but also with explicit representations in terms of the CSP process algebra.
5. If the explicitly defined processes were sufficiently simple to be implemented in a direct way, their compliance with the associated implicit proof obligations was shown using the proof rules [Sch00, pp. 197] for the satisfaction relation. If necessary, term re-writing based on the laws of the CSP process algebra was performed for the process, in order to reach a representation close enough to an implementation in the target programming language (Pascal) and operating system.
6. If the implementation of the sequential process P_i required more complex sequential algorithms the proof theories for nondeterministic sequential programs and distributed programs elaborated by Apt and Olderog [AO91] were applied, in order to show that the communication pattern used in the process, together with the sequential algorithms executed between communications, really implied the proof obligations P_i **sat** $S_i(tr, R)$.

The first five specification and verification techniques are all defined within the well-known denotational semantics and associated proof theories of "modern" CSP: Term re-writing based on the algebraic laws preserves failures equivalence, compositional proof rules about the satisfaction relation are defined for each CSP operator, and failures refinement preserves the satisfaction relation. The sixth technique, however, requires some explanation:

Verification of CSP With Sequential Imperative Program Parts. As is often the case in distributed systems design, the communication structure of the dual computer system and the sequential algorithms for queue management, serialisation of conflicting transactions, fault management and related activities were designed

separately. Since Pascal was used as programming language, it was only natural to use a conventional operational semantics interpretation and Hoare logic for reasoning about pre- and postconditions, in order to prove the correctness of the sequential parts. This left us with the task to prove that the effect of the sequential algorithms as visible on local process variables also implied the proof obligations P_i **sat** $S(tr, R)$ specified for the visible communication behaviour of each sequential process P_i. Though Hoare had indicated in [Hoa85, pp. 185] how to integrate local variables, assignment and control structures of imperative programming languages into sequential CSP processes, the validity of combining proofs obtained for sequential program fragments interpreted in an operational semantics with CSP process behaviour interpreted in the denotational models did not seem quite as obvious to us to be applied without further consideration.

To this end, the *distributed programs* introduced by Apt and Olderog [AO91] proved to be helpful: The authors consider networks $X = (S_1 \parallel \ldots \parallel S_n)$ of sequential processes

$$S_i \equiv S_{i0};\ \mathbf{do}\ \Box_{j=1}^{m_i} g_{ij} \rightarrow S_{ij}\ \mathbf{od} \quad (*)$$

In this representation, S_{i0}, S_{ij} are sequential nondeterministic program fragments written in an imperative style, operating on local variables [AO91, pp. 106]. Each g_{ij} is a communication construct of the form $g_{ij} \equiv B_{ij}\&c_{ij}!x$ or $g_{k\ell} \equiv B_{k\ell}\&c_{k\ell}?x$. The communication construct is structured into guards B_{ij} which are Boolean expressions over local variables and channels c_{ij} carrying messages $c_{ij}.x$. Channel outputs and inputs are denoted in the usual CSP style as $c_{ij}!x$ and $c_{k\ell}?x$. Communication between sequential processes S_i and S_k can take place over *matching* $g_{ij}, g_{k\ell}$ – that is, $g_{ij} = B_{ij}\&c_{ij}!x_{ij}$, $g_{k\ell} = B_{k\ell}\&c_{k\ell}?x_{k\ell}$ and $c_{ij} = c_{k\ell} = c$ – whenever both B_{ij} and $B_{k\ell}$ evaluate to *true* in the actual process states of S_i and S_k. The *effect* of the communication is equivalent to an assignment $x_{k\ell} := x_{ij}$ of the output variable value to the input variable.

The communication structure of the sequential processes S_i matched exactly with the communication pattern applied for the sequential processes of the dual computer system. Moreover, Apt and Olderog introduced an operational semantics for distributed programs which was identical with (nondeterministic) sequential program semantics for the sequential process parts S_{ij}. Indeed, it is shown in [AO91, pp. 334] that distributed programs X can be transformed into equivalent nondeterministic sequential programs $\nu(X)$, so that proofs about distributed programs can be performed using the rules for nondeterministic sequential program verification. Program $\nu(X)$ is given by

$$\begin{aligned}\nu(X) \equiv\ & S_{10};\ \ldots;\ S_{n0};\\ & \mathbf{do}\ \Box_{(i,j,k,\ell)\in\Gamma} B_{ij} \wedge B_{k\ell} \rightarrow x_{k\ell} := x_{ij};\ S_{ij};\ S_{k\ell};\ \mathbf{od}\end{aligned}$$

where the set Γ contains index quadruples of matching communication constructs $g_{ij} = B_{ij}\&c_{ij}!x_{ij}$ and $g_{k\ell} = B_{k\ell}\&c_{k\ell}?x_{k\ell}$ with $c_{ij} = c_{k\ell}$. For a channel c_{ij} let $\alpha(c_{ij})$ (the *channel alphabet*) denote the set of all pairs $c_{ij}.x$ with correctly

typed channel messages x. For distributed program X the alphabet $A = \alpha(X)$ is the union over all alphabets of channels referenced in X plus event $\checkmark$ indicating termination of X (if X terminates at all). Using abbreviation

$$r_\Gamma = A - \{c_{ij}.x_{ij} \mid (i,j,k,\ell) \in \Gamma \wedge B_{ij} \wedge B_{k\ell}\}$$

and augmenting $\nu(X)$ by fresh variables $tr_v : A^*$ and $R_v : \mathbb{P}(A)$ and corresponding assignments, we construct a new nondeterministic sequential program

$$\begin{array}{l} X' \equiv S_{10};\ \ldots;\ S_{n0};\ tr_v := <>;\ R_v := r_\Gamma; \\ \quad \textbf{do} \\ \quad\quad \Box_{(i,j,k,\ell)\in\Gamma} B_{ij} \wedge B_{k\ell} \rightarrow \\ \quad\quad\quad x_{k\ell} := x_{ij};\ tr_v := tr_v{}^\frown < c_{ij}.x_{ij} >;\ S_{ij};\ S_{k\ell};\ R_v := r_\Gamma; \\ \quad \textbf{od} \end{array}$$

which still has the same behaviour with respect to the local variables of $\nu(X)$ since the fresh variables tr_v, R_v are nowhere referenced within the sequential fragments S_{ij}. In [Pel97, pp. 87] we have constructed a syntactic mapping from the set of CSP processes X following the communication pattern (*) into the set of nondeterministic sequential programs structured like X'. Furthermore it has been shown that proof obligation X **sat** $S(tr, R)$ holds if and only if

$$\forall\, U : \mathbb{P}(R_v) \bullet S(tr_v, U)$$

is a **do**...**od** loop invariant of the associated sequential program X' and holds in case of termination. Intuitively speaking, the pair (tr_v, R_v) represents a failure of X, when interpreted in the denotational model, and R_v is a maximal refusal applicable in the current process state X/tr_v.

With this correspondence between CSP processes and nondeterministic sequential programs at hand, proof obligations X **sat** $S(tr, R)$ can be derived for the process by reasoning about its sequential program parts and local variables, using the operational semantics and Hoare logic.

Remarks and Related Publications. In [Pel91,Pel97,BCOP98] it is illustrated how this combination of sequential reasoning with algebraic, assertional and refinement verification methods can be applied, so that also projects of a larger scale, where a single verification technique would not suffice to discharge every type of proof obligation, can be effectively handled.

Observe that the necessity to embed description techniques for sequential algorithms into CSP has been realised by several authors, so that a variety of solutions is now available, allowing to pick the ones which are most appropriate for the description of each verification problem or for transformation of specifications into executable code: When using machine-readable CSP with the FDR tool, algorithms are specified using a functional programming language [Ros98, pp. 495]. The combined use of the functional programming paradigm and CSP has been

investigated extensively by several authors; we name Abdallah's work [Abd94] on the development of parallel algorithms based on functional specifications as an example. As an alternative to imperative and functional descriptions, the implicit specification of data manipulations has been made available by embedding Z or one of its object-oriented variants into CSP. For this combination, we refer to Fischer and Smith [FG97] and the literature cited there.

2.2 Formal Methods for the International Space Station

The International Space Station. The *International Space Station ISS*, launched in July 2000 and today still in its construction phase in orbit, is a joint venture between the United States, Russia, Japan, Canada and Europe. Managed through the European Space Agency ESA, Europe contributes to this huge international project by

- The Colombus Orbital Facility, a research laboratory to specialise in research into fluid physics, materials science and life sciences,
- The Automated Transfer Vehicle (ATV) to be used for carrying cargo from the earth to the ISS and – once docked at the ISS – for boosting the station higher in its orbit.

Moreover, Europe has developed and delivered several smaller sub-systems for the ISS. One of these is the *Data Management System DMS-R* for the Russian segment of the ISS. The main responsibilities of the DMS-R are guidance, navigation and control for the entire ISS, on-board system control, failure management and recovery, data acquisition and control for on-board systems and experiments. The DMS-R has been developed by ESA with an industrial team led by EADS ASTRIUM in Bremen, Germany.

The Fault-Tolerant Computer. The computing platform for the DMS-R is the *FTC*, a fault-tolerant computer system. Various fault-tolerant configurations can be selected for the FTC; the most reliable and at the same time the most complex one consists of a four-times redundant setting, where the four computer nodes cooperate according to Lamport's Byzantine Agreement Protocol [LSP82]. The corresponding FTC architecture is sketched in Figure 2.

The FTC communicates with other components in the ISS using a strictly synchronised frame protocol over redundant MIL-STD 1553 busses. Each FTC node has a layered hardware and software architecture: Applications are programmed in C and run on a ruggedised variant of the SUN Sparc CPU board developed by Matra in France. The application layer makes use of the Application Service Layer ASS for communication, time management and other services. The fault-tolerant mechanisms are encapsulated in the fault management layer FML. On each node, the FML is implemented on a separate hardware board equipped with transputer technology, with software programmed in OCCAM. Transputer links connect all FTC nodes with each other, in order to provide the communication infrastructure for the Byzantine Agreement Protocol. Data bus

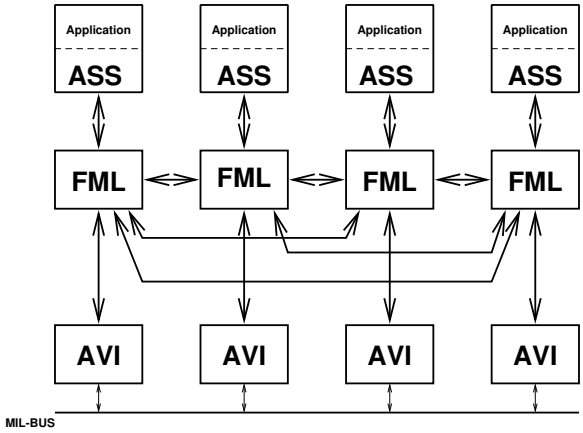

Fig. 2. FTC architecture.

access is managed through the avionics interface layer AVI, which again resides on a transputer board of its own and is also driven by OCCAM software.

Between 1995 and 1998 the author and his research team at the University of Bremen performed a variety of verification and testing activities for the FTC. Two major objectives from a wider list of correctness goals consisted in proving the absence of potential deadlocks or livelocks in the OCCAM code of the FML and AVI.

Verification Strategy for the FTC. The close relationship between OCCAM and CSP and the availability of the FDR model checker [For01], [Ros98, pp. 517] suggested an *abstract interpretation* approach for these tasks: OCCAM code P was mapped to a CSP abstraction $\mathcal{A}(P)$, thereby dropping all coding details of P without impact on communication behaviour. Then deadlock or livelock freedom was verified for $\mathcal{A}(P)$ by means of model checking with FDR. It soon became clear that model checking on CSP abstractions could only be used to verify small portions of OCCAM code; it would have been infeasible to map the approximately 24,000 lines of code as "one chunk" to CSP and then perform model checking on the complete system abstraction. Instead, a verification strategy combining several techniques had to be designed:

1. Abstraction methods were only used on small portions $P_1, \ldots, P_n$ of OCCAM code, resulting in a collection of CSP abstractions $\mathcal{A}(P_1), \ldots, \mathcal{A}(P_n)$.
2. Verification sub-goals were specified for the $\mathcal{A}(P_i)$ as refinement relations $SPEC_i \sqsubseteq \mathcal{A}(P_i)$ and verified via model checking for trace-, failures- or failures-divergence refinement. The choice of the semantic model depended on the sub-goal to be proved.
3. Compositional reasoning was used to derive the global verification goals from sub-goals verified for the $\mathcal{A}(P_1), \ldots, \mathcal{A}(P_n)$.

4. Generic theories were applied to re-use correctness results which only depend on generic characteristics of (sub-)systems: By showing that a CSP process $\mathcal{A}(P_i)$ complied with a specific communication design pattern (this proof was again performed by model checking) it was possible to use an instance of the generic theory, in order to prove that $\mathcal{A}(P_i)$ satisfied a desired property.

While the techniques 2. and 3. were just routine tasks, the abstraction techniques and the elaboration and application of generic theories required additional investigations.

Abstraction Techniques. The verification strategy sketched above implied that a variety of verification sub-goals would be investigated for OCCAM code portions $P_1, \ldots, P_n$. Furthermore, taking into account that verification goals on OCCAM level were expressed on a different syntactic level than the associated goals for CSP abstractions, a more formal definition of abstractions was required:

Definition 1. *Let P be an OCCAM or CSP process and p a property of P to be verified. Let $\mathcal{A}(P)$ denote a CSP process and $\mathcal{A}^*(p)$ a property defined on CSP level. Then the pair $(\mathcal{A}(P), \mathcal{A}^*(p))$ is called a valid abstraction for (P, p), if*

$$\mathcal{A}(P) \textit{ satisfies } \mathcal{A}^*(p) \quad \text{implies} \quad P \textit{ satisfies } p.$$

□

The most important abstraction technique applied was *abstraction through data independence*. Using this technique, the data ranges T of all OCCAM channel protocols and local process variables are partitioned into the minimal number of subsets $T_1 \cup \ldots \cup T_k = T$ which have to be distinguished in order to prove a given property p for an OCCAM process P. The CSP abstraction $\mathcal{A}(P)$ then operates on channels whose alphabet contains as many elements as partitions for T have to be distinguished; these can always be encoded as integral numbers $\{ 1, \ldots, k \}$. Control commands in P which are relevant for p and involve variables of type T are then abstracted to decisions on CSP level, where only the membership in a partition T_i is distinguished, but not the actual variable values itself.

Example 1. Suppose that channels c, d range over the natural numbers. We wish to prove that process system

$$\begin{aligned}
&\textit{channel } c, d : \mathbf{N} \\
&SYSTEM = (P \underset{\{|c|\}}{\parallel} Q) \\
&P = c!0 \rightarrow STOP \sqcap c!1 \rightarrow STOP \\
&Q = c?x \rightarrow (\mathbf{if}\ (x < 10)\ \mathbf{then}\ (d!0 \rightarrow STOP)\ \mathbf{else}\ (d!10 \rightarrow STOP))
\end{aligned}$$

is free of livelocks and always produces event $d.0$ before blocking. Formally, this property p can be expressed as $p \equiv ((d.0 \rightarrow STOP) \sqsubseteq_{FD} SYSTEM \setminus \{|\ c\ |\})$, $\sqsubseteq_{FD}$ denoting the failures-divergence refinement relation. Since the condition in

Q only depends on the two situations $x < 10$ and $x \geq 10$, it suffices to analyse the abstracted process system

$$
\begin{array}{l}
\mathit{channel}\ c', d' : \{\, 1, 2 \,\} \\
\mathcal{A}(\mathit{SYSTEM}) = (P' \underset{\{|c'|\}}{\parallel} Q') \\
P' = c'!1 \rightarrow \mathit{STOP} \\
Q' = c'?x \rightarrow (\textbf{if}\ \ (x == 1)\ \textbf{then}\ \ (d'!1 \rightarrow \mathit{STOP})\ \textbf{else}\ \ (d'!2 \rightarrow \mathit{STOP}))
\end{array}
$$

where channels c', d' are defined with the finite alphabet $\{\, 1, 2 \,\}$ instead of the infinite set $\mathbf{N}$; value 1 representing the partition $\{\, x < 10 \,\}$, value 2 partition $\{\, 10 \leq x \,\}$ of the original channels c, d. The abstracted property $\mathcal{A}^*(p)$ to be verified for $\mathcal{A}(\mathit{SYSTEM})$ is

$$\mathcal{A}^*(p) \equiv ((d'.1 \rightarrow \mathit{STOP}) \sqsubseteq_{FD} \mathcal{A}(\mathit{SYSTEM}) \setminus \{|\ c'\ |\})$$

referring to abstracted channels c', d'. □

With a CSP process $\mathcal{A}(P)$ generated from the original OCCAM process P by abstraction through data independence at hand, further simplifications could be made by constructing even more abstract CSP processes P' satisfying a refinement relation $P' \sqsubseteq \mathcal{A}(P)$ such that the desired property $\mathcal{A}^*(p)$ was preserved by this type of refinement. Then it sufficed to establish $\mathcal{A}^*(p)$ for P'. This technique is called *abstraction through refinement* and is less powerful, but considerably simpler than the abstraction through data independence, since it does not allow to further reduce the alphabet of the abstraction process P'.

Generic Theories. The local properties established for OCCAM processes $P_1, \ldots, P_n$ via abstraction and model checking had to be combined by compositional reasoning, in order to establish overall verification goals like deadlock and livelock freedom over given sets of input and output channels. In order to simplify this compositional reasoning process, *generic theories* were elaborated and applied in various situations, where different process sub-systems followed the same communication pattern, as required by the generic theory. For FTC verification, the generic parameters of each theory were

- number of processes involved,
- number and names of channels involved,
- specific parameters referring to re-occurring patterns in the communication behaviour.

Example 2. In the compositional reasoning process performed to prove deadlock freedom of the FML, it could be shown by model checking that each of the process sub-systems depicted in Figure 3 is a refinement of a process instance of type "multiplexer/concentrator" specified as *MUXCON* below. The definition of *MUXCON* is generic in the number N specifying how many outputs must be produced before the next input can be accepted and the number, names and

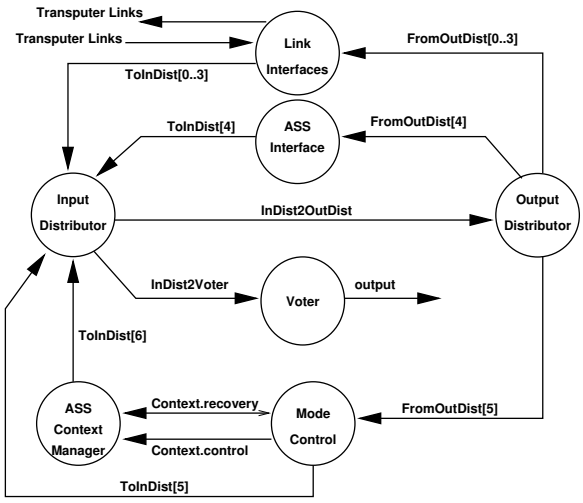

Fig. 3. Top-level processes of the fault management layer FML.

alphabet of input channels $\{in_1, \ldots, in_n\}$ and output channels $\{out_1, \ldots, out_m\}$. Observe that an instance of $MUXCON$ defined with $N = 0$ never refuses an input.

$$
\begin{array}{l}
MUXCON[N, \{in_1, \ldots, in_n\}, \{out_1, \ldots, out_m\}] = \\
\quad MC[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}](0)
\end{array}
$$

$$
\begin{array}{l}
MC[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}](n) = \\
\quad \textbf{if } (n = 0) \\
\quad \textbf{then } (GET[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}] \\
\qquad\qquad \Box\, (STOP \sqcap PUT[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}](1))) \\
\quad \textbf{else } PUT[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}](n)
\end{array}
$$

$$
\begin{array}{l}
GET[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}] = \\
\quad (\,\Box\, e : \{|\ in_1, \ldots, in_\ell\ |\} \bullet \\
\qquad e \rightarrow MC[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}](N))
\end{array}
$$

$$
\begin{array}{l}
PUT[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}](n) = \\
\quad (\,\sqcap\, e : \{|\ out_1, \ldots, out_m\ |\} \bullet \\
\qquad e \rightarrow MC[N, \{in_1, \ldots, in_\ell\}, \{out_1, \ldots, out_m\}](n-1))
\end{array}
$$

The following generic theory is associated with the above process class:

Theorem 1. *A network of process instances $P_1, \ldots, P_q$ from class $MUXCON[N, \{in_1, \ldots, in_n\}, \{out_1, \ldots, out_m\}]$ is free of deadlocks, if every communication cycle*

$$
\boxed{P_{j_1}} \xrightarrow{c_{j_1}} \boxed{P_{j_2}} \xrightarrow{c_{j_2}} \ldots \xrightarrow{c_{j_k-1}} \boxed{P_{j_k}} \xrightarrow{c_{j_k}} \boxed{P_{j_1}}
$$

contains at least one process instance $P_{j\ell}$ defined with $N = 0$.

It could be shown by model checking that for each communication cycle in the network of sub-systems shown in Figure 3, at least one sub-system is a refinement of a *MUXCON* instance with $N = 0$. Since deadlock freedom is preserved under refinement and refinement distributes through the parallel operator, this established deadlock freedom for the full FML layer. □

Remarks and Related Publications. The operation of the DMS-R system and its fault-tolerant computing platform has started with the launch of the Russian Service Module in July 2000 and is working nominally since then.

For more details about the ISS, the reader is referred to the web sites of the European Space Agency (http://www.esa.int) and of EADS (http://www.eads.net). A more comprehensive description of the verification activities[2] performed by the author and his research team for the International Space Station is given in [PB99]. Details about the fault-tolerant computer system FTC have been published in [UKP98]. The technical aspects of the FTC deadlock and livelock analysis are described in [BKPS97,BPS98]. The systematic application of generic theories and their mechanised verification with the HOL theorem prover has been sketched in [BCOP98]. Roscoe presents a detailed introduction and analysis of CSP abstraction concepts in [Ros98].

It should be noted that the abstractions from OCCAM to CSP which were required to prove absence of deadlocks or livelocks by model checking have been constructed in a manual way, relying on the verification engineers' expertise with respect to the decision whether an OCCAM code detail was relevant for communication behaviour or could be removed in the abstraction. Of course, this approach introduced the risk of inadvertently "losing" relevant code during the abstraction process. However, the activity of code abstraction differs considerably from the activity of code development itself. Moreover, the verification team was completely independent from the development team. Therefore we consider it as justified to assume that the probability of producing an abstraction error which exactly masks a programmed deadlock or livelock situation is low enough to be neglected. Observe finally, that the undecidability results presented by [LNR05] indicate that a mechanised abstraction may be generally infeasible, as soon as more complex data structures are involved.

2.3 Embedded Systems Testing for Airbus Avionic Systems

Testautomation Requirements Defined by Airbus. When Verified Systems International GmbH was founded as a spin-off company of the University of Bremen in 1998, the company received the first contract from Airbus for testing an avionics controller of the Airbus A340 aircraft.

The crucial requirements defined by Airbus in 1998 for the testing environment and its automation capabilities were

[2] These activities also included hardware-in-the-loop tests and statistical throughput analysis which have not been mentioned in the present contribution.

– Test data generation should be highly automated with respect to choices of data on individual interfaces, combinational patterns of input/output traces and their timing.
– All output interfaces of the system under test (SUT) should be continuously monitored, so that also transient output errors could be detected.
– Automated test oracles, that is, checkers of SUT responses against expected (i. e., specified) SUT behaviour should be simple to program and capable of detecting behavioural discrepancies with respect to interface data, causal chains of inputs and outputs, as well as timing.
– Regression testing should be fully automated.

Conventional Testing, as of 1998. At the time when Airbus defined the test automation requirements listed above, most conventional testing approaches used sequential test scripts: Each test execution consisted of an alternating sequence of

– inputs to the SUT,
– explicitly programmed checks of SUT responses against expected results.

This technique had considerable disadvantages with respect to the above mentioned test automation goals defined by Airbus: First, the simulation of components in the operational environment which interacted in parallel with the SUT were hard to express in sequential scripts, since all relevant interleavings of the parallel systems had to be programmed explicitly. This often led to over-simplified test scripts where SUT failures occurring only for special input/output sequences were overlooked. Second, illegal SUT outputs at event-based interfaces or illegal state changes at state-based interfaces were not detected if they occurred during the phase were the testing environment sent new inputs to the SUT: Checking was only performed at specific points in time, and often only at a subset of SUT output interfaces. Third, regression tests often failed though the SUT behaved correctly: This was caused by not considering all legal SUT output sequences in the test scripts. Instead, only one sequence was accepted as correct, which corresponded to the observed SUT behaviour in a certain revision. After changes in the SUT software, this output order changed slightly, but still legally. However, since the test script could only handle one sequence, the regression tests failed. Last, but not least, the effort for developing programmed test scripts was somewhat proportional to the length of the test execution: If different behaviours should be exercised on the SUT over a long period, all behavioural patterns had to be explicitly programmed, leading either to long and complex scripts or to over-simplified ones where the same pattern was executed over and over.

Specification-based Testing With CSP and RT-Tester. Based on the experiences with embedded systems testing for the International Space Station, Verified Systems' test automation tool RT-Tester had matured to a commercial product in 1998. In contrast to other approaches, test configurations were always designed as distributed systems, as indicated in Figure 4: The test automation tool offers

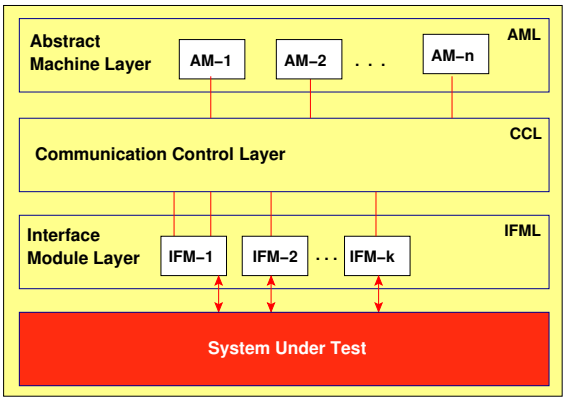

Fig. 4. Generic architecture of the RT-Tester test automation system.

the possibility to run an arbitrary number of concurrent *abstract machines (AM)* on the test engine, each machine performing one or more testing tasks like

- simulation of components in the operational environment of the SUT,
- stimulation of specific input sequences, in order to test a special test objective which can only be checked in a certain pre-state, that is, after a specific input trace starting with SUT initialisation,
- checking SUT outputs against expected results specifications.

The execution of abstract machines in hard real-time and the efficient communication between them is supported by multi threading mechanisms and a *communication control layer (CCL)* allowing to exchange data within a multi CPU/multi node cluster architecture for test engines (see Section 4). The CCL implements an abstract notion of communication channels. In the RT-Tester version available in 1998, this notion corresponded to the CSP channel concept implemented by the FDR tool, [Sch00, pp. 469]. Channels were used as the only means of communication between abstract machines[3]. In order to map the channel abstraction onto concrete SUT interfaces, *interface modules (IFM)* refined data from abstract channel events to concrete hardware driver calls or software interfaces and abstracted SUT outputs back to channel events. This abstraction concept allowed to re-use test specifications implemented as networks of AMs on different integration levels: If a software integration test accessed software interfaces $s_1, \ldots, s_n$ of the SUT software, and these mapped directly to hardware interfaces $h_1, \ldots, h_n$ in the integrated HW/SW system, then the abstract machines used in software the integration test could be re-used on HW/SW integration level, just by exchanging the interface modules.

[3] The current version of RT-Tester supports a broader variety of communication mechanisms: Test designers may combine the CSP channel concept described here with channels transporting structured C/C++ data and with shared memory data exchange.

The behaviour of abstract machines could either be programmed in C or – and this was considered as a major advantage when compared to other testing tools – specified in Timed CSP (TCSP). In order to use the syntax which is accepted by FDR, TCSP timing constructs had to be expressed as special events expressing the setting of a timer – that is, a clock counting from a given value $\delta > 0$ down to 0 – and indicating that the timer has elapsed: As shown by Meyer in his dissertation [Mey01], each network P of TCSP processes may be decomposed into parallel processes

$$P' = (P_U \underset{\{|s_0,\ldots,s_k,e_0,\ldots,e_k|\}}{\parallel} TIM) \setminus \{|\ s_0, \ldots, s_k, e_0, \ldots, e_k\ |\} \qquad (**)$$

such that

- P_U does not contain any timing operator like $WAIT\ t$ or $\overset{t}{\triangleright}$,
- $TIM = |||\, i : \{\, 0, \ldots, k \,\} \bullet T_i$ with timer processes

$$T_i = s_i?t \rightarrow ((\, WAIT\ t;\ e_i.t \rightarrow T_i) \,\Box\, T_i)$$

- P and P' are equivalent in the Timed Failures Model of TCSP.

Example 3. Consider the TCSP process network SYS with alphabet $A = \{\, in, out, a \,\}$,

$$\begin{aligned} SYS &= (P \underset{\{\, a \,\}}{\parallel} Q) \setminus \{\, a \,\} \\ P &= WAIT\ t;\ a \rightarrow in \rightarrow P \\ Q &= (a \rightarrow Q) \overset{u}{\triangleright} (out \rightarrow Q) \end{aligned}$$

This can be equivalently tranformed into

$$\begin{aligned} SYS' &= (((P' \underset{\{\, a \,\}}{\parallel} Q') \setminus \{\, a \,\}) \underset{\{|s_0,s_1,e_0,e_1|\}}{\parallel} (\, T_0 \,|||\, T_1)) \setminus \{|\ s_0, s_1, e_0, e_1\ |\} \\ P' &= s_0!t \rightarrow e_0.t \rightarrow a \rightarrow in \rightarrow P' \\ Q' &= s_1!u \rightarrow ((a \rightarrow Q') \,\Box\, (e_1.u \rightarrow out \rightarrow Q')) \end{aligned}$$

with T_0, T_1 as defined above. □

With this equivalence transformation at hand, the FDR tool can be used to translate TCSP specifications written with timer events s_i, e_i into transition graphs. In fact, only the P_U component of equation (**) has to be transformed: A transition graph interpreter which is part of RT-Tester and controls the AM execution handles the s_i, e_i events in real-time by setting timers of duration $s_i.t$ when this event is generated by the abstract machine and simulating an $e_i.t$-event as soon as time interval t has elapsed (Figure 5). This approach immediately solves the automated checking problem for timed traces by using an abstract machine performing a *back-to-back test* in parallel with the SUT during the test execution: The AM tracks every SUT input and output by navigating through the transition graph TG representing the CSP specification of the required SUT behaviour. Whenever an outgoing transition of the current state is

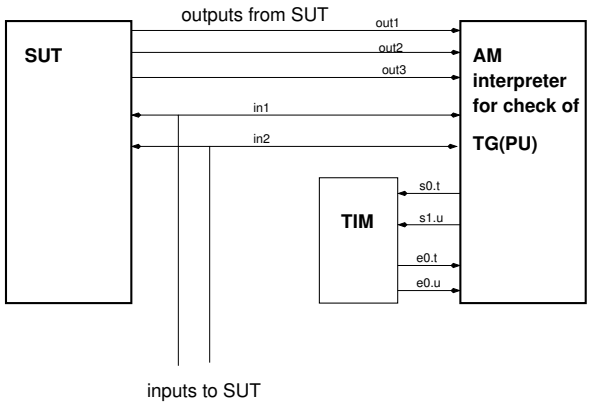

Fig. 5. Abstract machine acting as on-the-fly checker during a test execution.

labelled by a timer event $s_i.t$, the AM sets the corresponding timer for duration t. Whenever the timer signals an $e_i.t$-event and such a transition exists in the current state, it is taken by the AM. If the SUT produces an output *out* for which no transition exists in the *TG*-state marked by the AM as current, an output failure has been produced. The failure may be caused by a erroneous calculation of output data within the SUT or by generating an output too early. Since TCSP specification semantics assumes maximal progress, outputs which cannot be refused by the SUT must occur immediately, if not blocked by the environment.

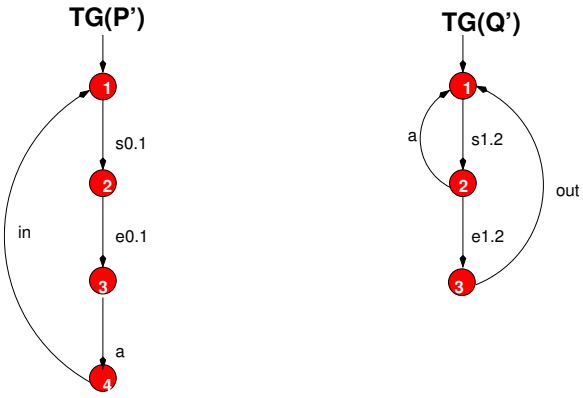

Fig. 6. Transition graphs of processes P', Q' from Example 3.

Example 4. Suppose the requirements for an SUT are expressed by TCSP process *SYS* as given in Example 3, with $t = 1$ and $u = 2$. Suppose further that

in is an input to the SUT and *out* an SUT output, while a cannot be observed during the test execution. The checking abstract machine interprets the transition graph depicted in Figure 7, which is the product graph generated from the representations for P' and Q' shown in Figure 6. Suppose the checker observes trace

$$< (3, in), (3, out), (4, out) >$$

Then a failure is detected at timed event $(4, out)$ because the checking AM assumes transition graph state 42 in Figure 7, after internally tracing

$$< (0, s_0.1), (0, s_1.2), (1, e_0.1), (1, \tau), (1, s_1.2), (3, e_1.2),$$
$$(3, in), (3, s_0.1), (3, out), (3, s_1.2), (4, e_0.1), (4, \tau), (4, s_1.2) >$$

and the *out*-event is not legal in this state. □

Observe that the checking technique sketched above only requires to specify SUT behaviour. If such a formal TCSP specification exists, no additional comparisons of observed test executions against expected results have to be programmed: Every SUT discrepancy is revealed by events for which no transitions exist in the current state of the checking AM.

On-the-fly checking has the further advantage that legal but nondeterministic SUT behaviour is not rejected by the checker, if the transition graph corresponds to a complete specification of legal SUT behaviour. If unnormalised transition graphs are used as in Figure 7, the checker has to mark several possible states as current. The SUT behaviour is accepted as long as at least one possible state exists. This procedure only allows checking in soft real-time, since the decision whether SUT behaviour is legal now not only depends on the maximal number of outgoing transitions to be compared against the observed behaviour but also on the number of possible states. Therefore, if hard real-time checking is required, normalised transition graphs should be used. These can also be generated by FDR.

The transition graph interpretation technique sketched above is obviously also suitable for real-time simulation: Abstract machines now operate on transition graphs representing environment specifications and generate events which are inputs to other AMs or to the SUT and choose different paths through the transition graph in places where several events are possible. This technique is also the starting point for systematic test data generation which will be discussed in more detail in Section 3.

Remarks and Related Publications. The testing activities performed by Verified Systems for Airbus could be extended since the first contract in 1998, so that today our testing projects comprise the A318, A340-500/600 and A380 aircrafts. The tested controllers are

– the Cabin Communication Data System CIDS developed by Airbus KID-Systeme,

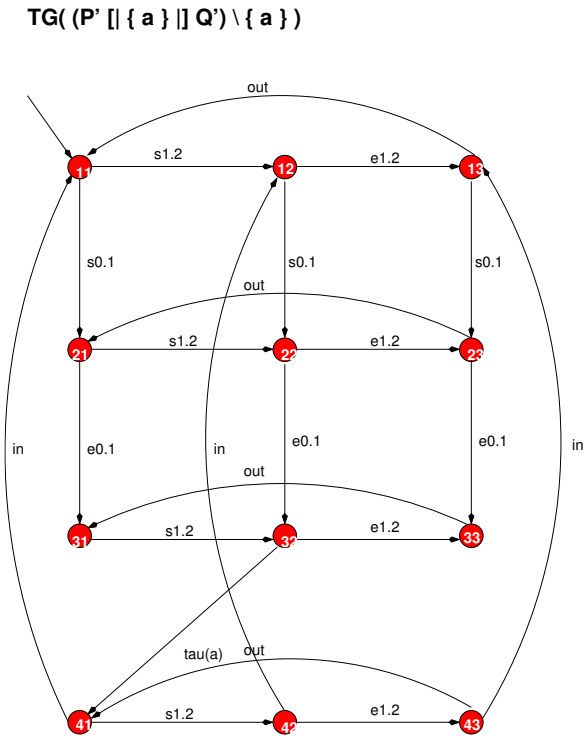

Fig. 7. Transition graph for parallel composition of P', Q' from Example 3.

– the Smoke Detection Controller SDF, also developed by Airbus KID-Systeme,
– the Integrated Modular Avionics (IMA) Modules, a new type of controllers which is first used for the Airbus A380 and has been developed by Diehl Avionics (Germany) in cooperation with Thales (France).

The testing activities comprise

– hardware/software integration tests for one or more controllers with integrated system and application software,
– so-called bare-module tests, where controller hardware, firmware, operating system and configuration data are tested before integration of the application software,
– software integration tests performed both on PC host simulation environments and on target hardware,
– module tests.

The same CSP-based testing technology has been applied by the author, his research group at Bremen University and by Verified Systems in several other projects: (1) Tests for a tramway crossing control system developed by EL-PRO in Berlin, (2) Tests for an aerospace satellite controller developed by OHB

in Bremen [SMH99], (3) tests for interlocking system components developed by Siemens Transportation Systems in Braunschweig, (4) UNISIG conformance tests for radio-based train control systems developed by Siemens Transportation Systems in Berlin[4] and (5) tests of automotive controllers for DaimlerChrysler in Sindelfingen.

For untimed process algebras, the relationship between semantic equivalence (or refinement) and testing has first been observed by Hennessy and De Nicola [DNH84], within the context of Hennessy's Acceptance Tree semantics which corresponds to the failures model of CSP. Brinksma observed that these theoretic results could be applied in practice and developed techniques for automated conformance testing based on LOTOS specifications; an extensive bibliography is given in [BT00]. The analogous concepts and further improvements have been elaborated for the untimed CSP world by the author in collaboration with Michael Siegel in [PS96,Pel96,PS97,Pel97].

During the period 2000 — 2004, our research activities related to real-time testing of avionics systems have been performed within the European research project VICTORIA[5] which focused on novel architectures, development and verification technologies for aircraft electronic systems. The test concepts developed for tests of integrated modular avionics have been described in more detail in [Pel03,MTB+04]. The advantages of interface abstraction and the resulting possibility to re-use test specifications on different integration levels have been discussed in more detail in [PT02]. The algorithm implemented in the RT-Tester system for automated checking of timed traces observed during test executions against SUT specifications has been published in [Pel02]. Further details about automated testing against Timed CSP specifications, in particular automated test data generation, are described below in Section 3.

3 Specification-Based Hard Real-Time Testing – Test Automation for Timed CSP

Solved Problems. It has already been sketched in Section 2.3 how the *test oracle* problem – that is, automated checking of SUT behaviour against expected results – and the simulation problem are solved for specification-based testing in a TCSP context: Using Meyer's structural decomposition of TCSP process networks, checking can be performed in back-to-back manner on transition graphs as the one depicted in Figure 7, and simulation can be performed by abstract machines deriving their behaviour from paths through these graphs.

[4] The UNISIG standard has been defined for train control and communication within the European Train Control Systems (ETCS) initiative. The test automation concept has been published in [Ken04].

[5] Detailed project descriptions are available under web links http://www.informatik.uni-bremen.de/agbs/projects/victoria/ and http://www.euproject-victoria.org/.

Test Data Generation. For automated test data generation, that is, generation of timed traces containing inputs to the SUT, the evaluation of transition graphs has to be further refined. The reason for this is that graphs like the one depicted in Figure 7 do not encode information about the relative durations when several active timers will elapse. As a consequence, the graph suggests elapsed-timer transitions $e_i.t_i$ which cannot occur in a certain state, because another timer $e_k.t_k$ will elapse before.

Example 5. When initially entering state 22 in Figure 7, transition $e_1.2$ cannot occur, since timer $e_0.1$ will elapse before.

To solve this problem, we analyse an extended class of transition graphs, where each state is also annotated by the vector $(u_0, \ldots, u_k) \in (\mathbb{R}_0^+)^k$ of time durations left for each timer until it elapses.

Example 6. When initially entering state 22 in Figure 7, the extended transition graph has encoding $(22, (1, 2))$ for the corresponding state, since the $e_0.1$ event will occur after 1 time unit, and it would take 2 time units to elapse, before the $e_1.2$ event could occur.

The transition system represented by these extended transition graphs has uncountably many states, but – adapting the concept of *regions* which has been introduced for Timed Automata [SVD01] – it can be abstracted to another graph G_A which only shows the finitely many different timer constellations which influence whether visible transitions are enabled or not. Graph G_A is generated from the original graph as the one shown in Figure 7 by repeated transformations with the goal to identify all nodes which are equivalent in this sense.

Example 7. To illustrate this process Figure 8 shows the initial part of G_A, as it results from this transformation applied to the graph from Figure 7.

In Figure 8, states like $(22, (1, 2))$ are identified with $(21, (1, 0))$, because in any case timer $e_0.1$ will elapse first. Original state 42 is now partitioned into 4 regions, each determining a different class of future behaviour: In $(42, (0, 2))$ only an *in* event can occur and after that, the future behaviour is determined by node $(11, (0, 0))$[6]. This behaviour does not change until 1 time unit has passed after entering $(42, (0, 2))$, which is marked by the transition 1 to state $(42, (0, 1))$: If an *in* event occurs exactly in this state, both timers will elapse simultaneously in state $(32, (0, 0))$, leading to a new type of behaviour, because a nondeterministic decision to engage into *out* instead of the hidden a event is possible. When state $(42, (0, 0.5))$ is reached, the hidden a transition can never occur in the next execution round, because the $e_1.2$ timer always elapses before $e_0.1$. Finally, when state $(43, (0, 0))$ is reached, both *in* and *out* become enabled.

For the resulting region graphs G_A it is possible to adapt a result by Chow [Cho78] which was established for testing untimed automata: Using a specific strategy to cover the graph by test traces and by checking that the SUT

[6] Note that in our notation $(42, (0, 2))$ identifies all nodes $(42, (0, 2-d))$ with $0 \leq d < 1$. The analogous applies to $(42, (0, 1))$ and $(42, (0, 0.5))$.

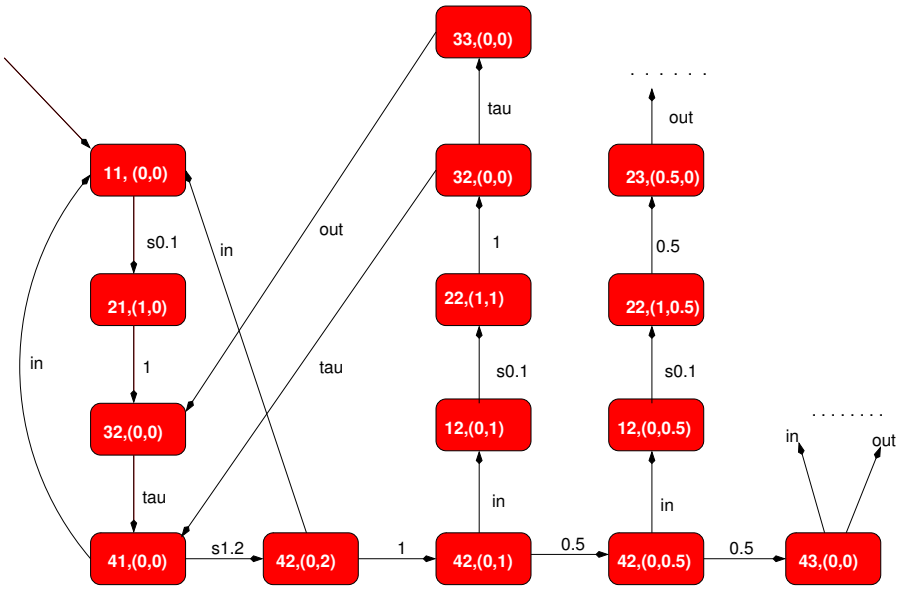

Fig. 8. Partial structure of graph G_A transformed from Figure 7.

really reaches the intended target states in each test it is possible to *prove* (or disprove) a failures refinement relation between TCSP specification and SUT by executing a finite number of test traces. To this end, a *finite variability assumption* has to be stated for the SUT, saying that transient error states must be stable for at least a minimal time duration $\delta > 0$. This assumption is realistic from a practical point of view, since controllers cannot act arbitrarily fast.

Remarks and Related Publications. The simpler nature of the TCSP test data generation problem, when compared to the solution for Timed Automata has an important practical implication: In [SVD01] the authors emphasise that the number of traces to be generated for exhaustive testing of Timed Automata will be so large for non-trivial problems, that a complete execution of the corresponding tests would be infeasible. Moreover, this also indicates that a heuristic selection of "useful" test cases from the complete set might be equally infeasible, or at least extremely complicated, since there are too few known criteria for distinguishing "important" test traces from "less important" ones. We expect that for the TCSP approach a much smaller number of test traces will be required for exhaustive testing of non-trivial systems whose behaviour is required to be a timed trace refinement of a TCSP specification. The main reason for this assumption lies in the fact that TCSP does not allow to refer to clock values and durations in an explicit way within boolean expressions: Time is only "experienced" by observing that certain events occur before or after timers have elapsed.

It is interesting to note that Schneider has introduced a new operator in [Sch00, p. 272], the *timed event prefix* $a@u \rightarrow Q(u)$. This operator allows to measure the time u which has passed between offering an event a to its environment and the actual occurrence of the event. The measurement u may be used as free variable in CSP process terms, in particular, it may appear in communication guards. This operator is not allowed in our solution, where the timeout $\overset{t}{\triangleright}$ and related syntactic abbreviations like $WAIT\ t$ are the only admissible time-related operators. Since the timed prefix offers the possibility to access duration values explicitly within a TCSP specification, we expect that the TCSP test data generation problem will become as complex as the one for Timed Automata, if this operator is also admitted.

Further references related to specification-based testing have already been given in Section 2.3.

4 Executable Formal Specifications: The Hybrid Low-Level Language Framework

From Timed CSP to Specification Formalisms for Hybrid Systems. The application domains where most of our verification activities described in Section 2 were performed typically evaluated and acted on physical parameters of discrete (e. g., states of signals and points) and time-continuous (e. g., temperature, speed, thrust) nature. This led to the investigation of hybrid specification formalisms, where not only time-discrete changes of variables, but also time-continuous evolutions of "analog" parameters could be described.

After the successful applications of TCSP a natural candidate for such a formalism was He Jifeng's *Hybrid CSP (HCSP)* extension [Jif94]. Therefore its applicability was investigated by Amthor [Amt99] with respect to test automation problems. However, it turned out that the restriction of time-continuous evolutions to local CSP process variables was too severe to be used above software design level: In physicals, global time-continuous observables occur naturally as variables of physical laws and models. In contrast to this, HCSP already operates on a discretised level: The actual state of time-continuous evolutions specified in for local HCSP process variables can only be communicated to other observers by using CSP channels at discrete points in time. Therefore, according to our assessment, HCSP is well-suited for the software design of hybrid systems, but less usable for physical modelling.

As an alternative to HCSP, Henzinger's Hybrid Automata [Hen96] combine synchronous CSP-style communication with global time-continuous variables. Each sequential automaton concurrently acts on these analog parameters according to flow equations, that is, differential equations describing the continuous evolution of global parameters. Discrete changes can be triggered during transitions between control states. While offering the basic tools for physical modelling, Hybrid Automata are designed as flat networks, so that their practical application for large systems leads to specifications which are not sufficiently

structured and therefore unmanageable. The hierarchic extension of Hybrid Automata developed by Alur et. al. [ADE+01] does not incorporate events in their semantic model; communication between parallel components is only performed via shared variables. Moreover, the syntactic representation is rather specialised, so that the chance for wider acceptance in an industrial application context seems rather low.

Inspired by Hierarchical Hybrid Automata and with the intention to reach a wider industrial community, we therefore decided to design a hybrid extension of the UML. To this end, the UML2.0 profile mechanism offers a rather well-defined means to extend the existing UML formalisms by new features and assign meaning to them. The resulting *HybridUML* profile has been published in [BBHP03], further description of its semantics can be found in [BBHP04]. HybridUML extends UML Statecharts [RJB99] by description mechanisms for invariants and flow conditions which hold for the complete hierarchy of states subordinate to the one where they have been defined (Figure 9). Additionally, class diagrams are used to model state and sequential operations, and architectural aspects can be specified using the structure diagrams.

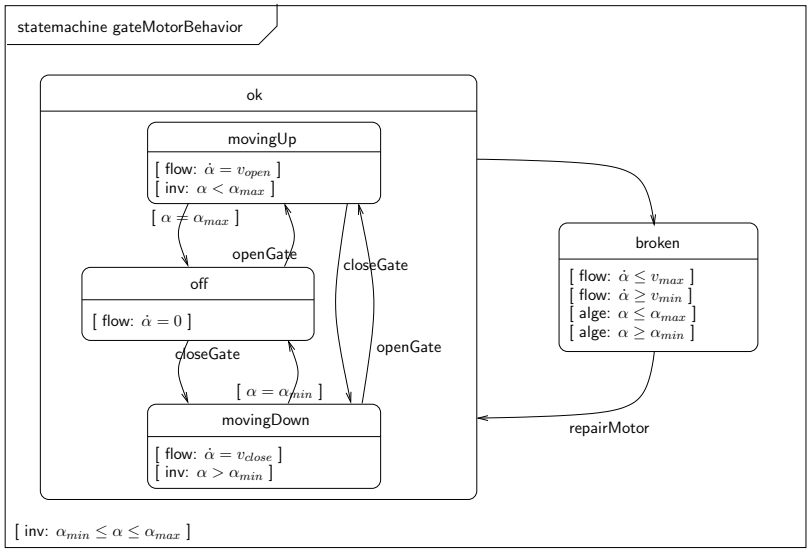

Fig. 9. HybridUML Statechart with invariants and flow conditions.

The Hybrid Low-Level Language Framework. Apart from formal verification of HybridUML specifications, a major research objective is to provide the means to execute the specifications in hard real-time, for the purpose of application development, simulation and testing. To this end, the HL^3 framework developed within our research group in cooperation with Verified Systems consists of a re-

usable hard real-time runtime environment R and a design pattern P for compilation targets of arbitrary hybrid specifications. Given a high-level formalism H – such as HybridUML – for the description of hybrid systems, transformations Φ_H from high-level specifications S into instances $\Phi_H(S)$ of the HL^3 pattern P can be developed. For $(\Phi_H(S), R)$, a formal semantics $\mathcal{S}(\Phi_H(S), R)$ based on timed state-transition systems is defined so that the transformation both provides a semantic definition of S and an executable program whose behavior will be consistent with $\mathcal{S}(\Phi_H(S), R)$. Similar to machine code, HL^3 should not be used for manual programming, but as a target language for automated transformations. In contrast to machine code, the real-time semantics of HL^3 program can be determined in a direct way, thereby assigning formal meaning to the high-level specification used as the transformation source. This is achieved by using a very limited range of instructions for multi-threading, timing control, and consistent handling of global state in presence of concurrency.

Remarks and Related Publications. The transition from CSP-based formalisms to hybrid systems has been performed since 1999 within the research project HYBRIS[7] Currently, our main research focus lies on the elaboration of a testing theory for HybridUML. An instance of the HL^3 framework is already used in the latest version of the RT-Tester tool, and test support for specification-based testing against HybridUML is currently developed. On a more application-oriented level, the HL^3 framework is instantiated for novel application domains, in particular for railway control systems [HP03].

The importance of semantically well-defined real-time execution environments has also been noted by other authors: For the Duration Calculus [ZRH93], implementable subsets have been investigated by several authors, see Ravn [Rav95] and further references given there. Our HL^3 framework competes with Henzinger's Giotto [HHK03] which can be used for implementing executable Hybrid Automata and similar high-level formalisms. Giotto and HL^3 are similar with respect to the time-triggered scheduling of discretised time-continuous control functions. Our approach is slightly broader than Giotto, since it aims at creating both executable target applications and/or the corresponding testing environments, allows to distinguish between discretised time-continuous and discrete control functions in an explicit way and offers the mechanisms for implementing both CSP-style interleaving semantics and synchronous "true parallelism" semantics as required for executable Statecharts and synchronous languages.

5 Conclusion

In this article, the application of CSP in industrial projects has been described. The projects focused on verification and test of embedded real-time systems

[7] Efficient Analysis of Hybrid Systems HYBRIS. Research project within the priority programme *Software Specification – Integration of Software Specification Techniques for Applications in Engineering* initiated by the Deutsche Forschungsgemeinschaft DFG. Information available under http://tfs.cs.tu-berlin.de/projekte/indspec/SPP.

in the avionics and space application domains. References to further projects from these and other domains were given. The challenge of large-scale project verification and test led to numerous research activities – some of which have been sketched in this article – which were motivated by the need to combine various methods in order to cope with the size and complexity of the problems involved.

Since 1998, more than twenty verification engineers from Verified Systems International GmbH, the University of Bremen and from our customers have been involved in the CSP-based verification activities mentioned in this article. From our point of view this is a sufficient proof for the applicability of Formal Methods in general and, in particular, CSP. The other, even more important, benchmark for the success of verification efforts is certainly the number of problems identified in the analysed systems. Readers will understand that an explicit mentioning of "interesting" errors and their severity is not in the interest of our cooperation partners. However, it can be said that the databases managed by Verified Systems to document discrepancies uncovered during verification and testing projects since 1998 contain about 2000 entries. About one third of these findings are a direct consequence of Formal Methods applications in verifications or automated tests.

On the other hand, the search for novel formal description techniques which might appeal to a wider group of software or system developers is certainly not completed and still sub-divided into many competing and sometimes incompatible approaches.

According to our experience, the acceptance of a formal specification technique is considerably increased when specifications can be executed at least on simulation or prototyping level. The return of investment gained by uncovered bugs and discrepancies during verification, validation and testing does not seem to be a sufficient motivation to learn and use a formal description technique and associated methods and techniques in addition to the programming languages used to implement the executable system. We are therefore convinced that the most promising strategies for formal methods in industry are based on model-based development, where specifications can be directly transformed into efficient executable code. These observations have led to the research work on low-level formalisms which are suitable compilation targets for various more abstract specification languages and can be executed in hard real-time, as described in Section 4.

Finally, we would like to emphasise that the application of Formal Methods – in particular in the field of safety-critical systems – should always be considered as one means in a collection of several others which together ensure the quality and dependability of the products we develop. This collection also comprises techniques, tools and skills which are far less challenging from a scientific perspective (think of reliable configuration management, error reporting, project budget management, ...) but contribute to the overall product quality, just as our latest advances in formal verification.

Acknowledgements. I would like to express my gratitude to the organisers and speakers of the *25 Years of CSP* event at the London South Bank University, for creating a stimulating conference with numerous interesting – sometimes even exciting – contributions and discussions. My special thanks go to Ali Abdallah for organising this outstanding event and for expertly compiling the conference proceedings.

References

[Abd94] A.E. Abdallah. Derivation of Parallel Algorithms: From Functional Specifications to csp Processes. In B. Moller, editor, *Proceedings of Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 67–96. Springer-Verlag, August 1994.

[ADE+01] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. *Lecture Notes in Computer Science*, 2211:14–31, 2001.

[Amt99] P. Amthor. *Structural Decomposition of Hybrid Systems – Test Automation for Hybrid Reactive Systems.* Monographs of the Bremen Institute of Safe Systems (BISS) No. 13, University of Bremen, October 1999.

[AO91] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs.* Texts and monographs in computer science. Springer, 1991.

[BBHP03] Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska. HybridUML Profile for UML 2.0. SVERTS Workshop at the ⟨⟨ UML ⟩⟩ 2003 Conference, October 2003. http://www-verimag.imag.fr/EVENTS/2003/SVERTS/.

[BBHP04] K. Berkenkötter, S. Bisanz, U. Hannemann, and J. Peleska. Executable HybridUML and its application to train control systems. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*, pages 145–173. German Research Foundation DFG, Springer, 2004.

[BCOP98] B. Buth, R. Cardell-Oliver, and J. Peleska. Combining tools for the verification of fault-tolerant systems. In B. Buth, R. Berghammer, and J. Peleska, editors, *Tools for System Development and Verification*, volume 1 of *Monographs of the Bremen Institute of Safe Systems*, pages 41–69. Shaker, 1998.

[BKPS97] B. Buth, M. Kouvaras, J. Peleska, and H. Shi. Deadlock analysis for a fault-tolerant system. In M. Johnson, editor, *Algebraic Methodology and Software Technology. Proceedings of the AMAST'97, Sidney, Australia, December 1997*, volume 1349 of *LNCS*, pages 60–75. Springer, December 1997.

[BPS98] B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology. Proceedings of the 7th International Conference, AMAST 98, Amazonia, Brazil, January 1999*, volume 1548 of *LNCS*, pages 124–139. Springer, January 1998.

[BT00] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Proceedings of Summer School MOVEP'2k Modelling and Verification of Parallel Processes*, pages 44–50, Nantes, July 2000.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.

[DNH84] R. De Nicola and M. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[FG97] C. Fischer and Smith G. Combining CSP and Object-Z: Finite trace or infinite trace semantics? In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Verification, and Testing (FORTE/PSTV'97)*, pages 503–518. Chapman & Hall, 1997.

[For01] Formal Systems (Europe) Ltd. *Failures–Divergence Refinement – FDR2 User Manual*, 2001. http://www.formal.demon.co.uk/FDR2.html.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[HHK03] Th. A. Henzinger, B. Horowitz, and Chr. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[HP03] A. E. Haxthausen and J. Peleska. Generation of executable railway control components from domain-specific descriptions. In G. Tarnai and E. Schnieder, editors, *Formal Methods for Railway Operation and Control Systems: Proceedings of Symposium FORMS*, pages 83–90, Budapest, May 2003. L'Harmattan Hongrie.

[JH87] He Jifeng and C. A. R. Hoare. Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing*, 2:1–12, 1987.

[Jif94] He Jifeng. From CSP to hybrid systems. In A.W. Roscoe, editor, *A Classical Mind, Essays in Honour of C.A.R. Hoare*, International Series in Computer Science, pages 171–189. Prentice Hall, 1994.

[Ken04] D. Kendelbacher. *Architekturkonzept und Designaspekte einer signaltechnisch nichtsicheren Kommunikationsplattform für sicherheitsrelevante Bahnanwendungen*. PhD thesis, University of Bremen, Department of Mathematics and Computer Science, 2004. Available under http://elib.suub.uni-bremen.de/publications/dissertations/E-Diss835_dis_50b.pdf.

[LNR05] R. Lazic, T. Newcomb, and B. Roscoe. On model checking data-independent systems with arrays with whole-array operations. In A. Abdallah, C. B. Jones, and J. W. Sanders, editors, *Twenty-five Years of Communicating Sequential Processes*, LNCS. To appear, Springer, 2005.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[Mey01] O. Meyer. *Structural Decomposition of Timed-CSP and its Application in Real-Time Testing*. PhD thesis, TZI Center for Computing Technologies, University of Bremen, Germany, 2001.

[MTB+04] O. Meyer, A. Tsiolakis, S.-O. Berkhahn, J. Kruse, and D. Martinen. Automated testing of aircraft controller modules. In *Proceeding s of the 5th International Conference on Software Testing ICSTEST*, Düsseldorf, April 2004. SQS. Extended abstract and slides available under http://www.informatik.uni-bremen.de/∼tsio/papers/.

[PB99] J. Peleska and B. Buth. Formal Methods for the International Space Station ISS. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Re-*

cent Insights and Avances, number 1710 in LNCS State–of–the–Art Survey, pages 363–389. Springer, 1999.

[Pel91] Jan Peleska. Design and verification of fault tolerant systems with csp. *Distributed Computing*, 5(1):95–106, 1991.

[Pel96] J. Peleska. Test automation for safety-critical systems: Industrial application and future developments. In M.-C. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 39–59, Berlin, Heidelberg, New York, 1996. Springer-Verlag.

[Pel97] J. Peleska. *Formal Methods and the Development of Dependable Systems*. Bericht Nr. 9612. Christian-Albrechts-Universität Kiel, Institut für Informatik und praktische Mathematik, 1997. Habilitation thesis, available under http://www.informatik.uni-bremen.de/agbs/jp.

[Pel02] Jan Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *Proc. of the Sixth Biennial World Conference on Integrated Design & Process Technology (IDPT2002)*. Society for Design and Process Science, June 2002. Available under http://www.informatik.uni-bremen.de/agbs/jp/papers.

[Pel03] J. Peleska. Automated testsuites for modern aircraft controllers. In R. Drechsler, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 1–10, Aachen, 2003. Shaker.

[PS96] J. Peleska and M. Siegel. From Testing Theory to Test Driver Implementation. In M.-C. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 538–556, Berlin, Heidelberg, New York, 1996. Springer-Verlag.

[PS97] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, 1997.

[PT02] J. Peleska and A. Tsiolakis. Automated Integration Testing for Avionics Systems. In *Proceedings of the 3rd ICSTEST – International Conference on Software Testing*, April 2002. Extended abstract and slides available under http://www.informatik.uni-bremen.de/agbs/jp/papers/ftrtft98.ps.

[Rav95] A. P. Ravn. Design of embedded real-time computing systems. Technical Report ID-TR 1995-170, ID/DTU, Lyngby, Denmark, October 1995. dr. techn. dissertation.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference Manual*. Addison-Wesley, 1999.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[Sch00] S. Schneider. *Concurrent and Real-time Systems – The CSP Approach*. Wiley and Sons Ltd., 2000.

[SMH99] H. Schlingloff, O. Meyer, and Th. Hülsing. Correctness Analysis of an Embedded Controller. In *Proceedings of DASIA (Data Systems in Aerospace) '99 Conference*, volume ESA SP-447, Lisbon, Portugal, 1999.

[SVD01] Jan Springintveld, Frits W. Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, 2001.

[UKP98] G. Urban, H.-J. Kolinowitz, and J. Peleska. A survivable avionics system for space applications. In *The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, FTCS-28, Munich, Germany, June 23-25, 1998*, pages 372–379. IEEE Computer Society, June 1998.

[ZRH93] Chaochen Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In *Hybrid Systems*, pages 36–59. The Computer Society of the IEEE, 1993. Extended abstract.