Deadlock Analysis for a Fault-Tolerant System

Bettina Buth*, Michel Kouvaras**, Jan Peleska***, and Hui Shi†

Abstract. This article presents an approach for the verification of communication properties in large-scale real-world embedded systems by means of formal methods. It is illustrated by examples and results obtained during an industrial verification project performed for a fault-tolerant system designed and implemented by Daimler-Benz Aerospace for the International Space Station ISS. The approach is based on CSP specifications and the model-checking tool FDR. The task is split into manageable subtasks by applying an abstraction technique for restricting the specifications to the essential communication behaviour, modularization according to the process structure, and a set of generic theories developed for the application.

1 Introduction

One of the essential obstacles for the acceptance of formal methods during the last years is their failure to scale up to realistic applications. In our experience this problem can only be overcome by a combination of methods and the use of suitable tools that are optimized for the efficient handling of well-defined subtasks. This article reports work performed over the last 1.5 years with an overall effort of about one man year which supports the above claims.

JP Software-Consulting in collaboration with the Bremen Institute for Safe System (BISS) were contracted by Daimler-Benz Aerospace (DASA) to perform an analysis of a fault-tolerant data management system for the International Space Station (ISS). The software was made available in form of occam pseudo code resp. occam code [4]. The main concern of the analysis described in this article was to ensure freedom from deadlock with regard to process communication. This analysis is the first layer of a full verification suite ranging from formal software verification of various correctness aspects to hardware-in-the-loop tests of the complete system.

The strong relation between occam and CSP as well as previous experiences with CSP and the model checker FDR suggested the following procedure:

- derive a CSP abstraction from the occam (pseudo) code

* BISS, Bremer Institute for Safe Systems, EMail bb@informatik.uni-bremen.de
** Daimler-Benz Aerospace, Bremen
*** JP Software-Consulting, Bremen, EMail jp@informatik.uni-bremen.de
† University of Bremen, EMail shi@informatik.uni-bremen.de

– use FDR to check that the abstraction is a refinement of a deadlock free process DF.

First studies soon showed that though in essence a sound approach it could not be used without further considerations. The main obstacle was the size of the software to be analyzed. Furthermore, the result "system is free of deadlocks" is too coarse to be of practical value. Instead more refined proof obligations like "none of the processes in the system will continuously block communication on their relevant input channels" should be investigated. A true abstraction catching all details of the given code would be by far too large to be submitted to the model checker in one piece: the state space would explode. This report describes our methodology how to overcome these problems by using abstraction, the compositional theory of CSP, algebraic transformations, generic theories, and last but not least refinement checks with the tool FDR in combination. The approach is illustrated by examples from the verification of the Fault Management Layer (FML) of the system which is responsible for all aspects of fault-tolerance in the system. In the following section an overview of the FML is given together with a more detailed description of the goal of the analysis. Section 3 contains information about the methods employed for the verification and their respective rôle. The technical results are summarized in Section 4, followed by a collection of experiences and goals for future work in the conclusion.

2 Technical Background: The Fault Management Layer

2.1 The Fault-Tolerant Computer

The FML software to be analyzed is part of a fault tolerant computer to be used in the International Space Station (ISS) to control space station assembly, reboost operations for flight control, and data management for experiments carried out in the space station. In order to understand the rôle of the FML an overview of the overall structure of this fault-tolerant computer is necessary.

The overall architecture consists of up to four communicating lanes, each providing services for the applications. Each of these lanes is structured into an application services layer (ASS), a fault management layer (FML), and the avionics interface (AVI). The ASS resides on the application layer board and contains table driven services for the application software and the operating system. The AVI is in charge of the MIL Bus protocol handling according to predefined timing slot allocations. These are defined in an input/output table. The FML is inserted between ASS and AVI, and is responsible for the detection of data failures and the isolation and reintegration of lanes. In each lane, the application layer plus ASS runs on a customized Matra board using a SPARC CPU. Both FML and AVI reside on separate transputer boards. The lanes communicate only at FML level using the transputer links. Each FML uses up to three links for communication with the other lanes, and one link (link 0) for communication with the AVI. Data transfer with the ASS is performed using a VME interface. See Figure 1 for the architecture of a full four-lane system.

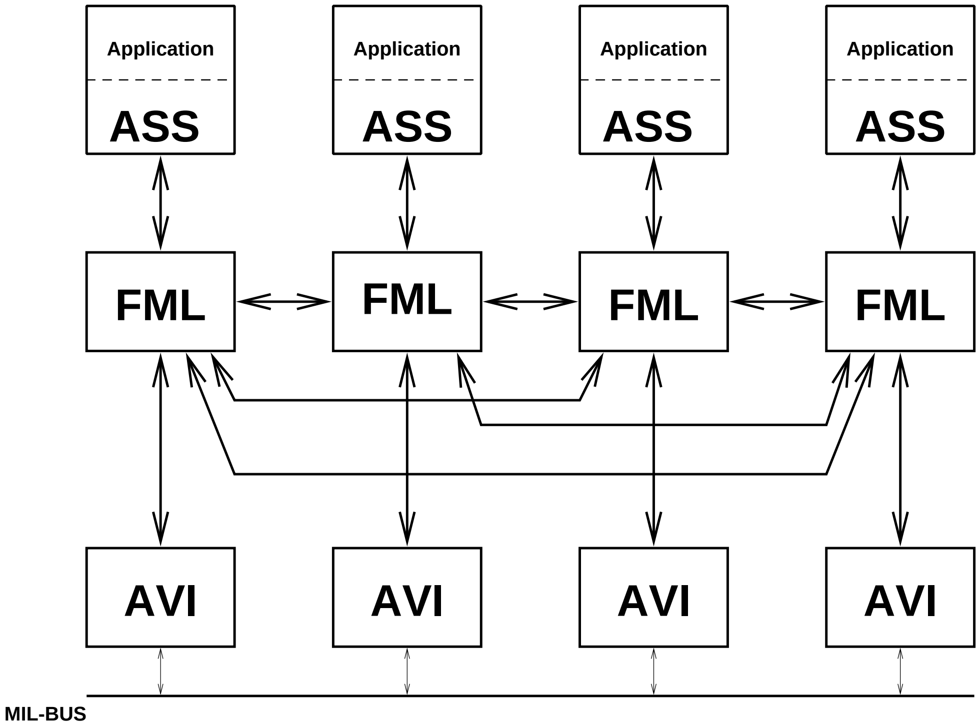

Figure 1. FTC architecure

The function of the FML is twofold: First, it provides the interface between the ASS and AVI of one lane, transferring messages from AVI to ASS and vice versa. Second, it performs the data transfer between lanes thus allowing communication between the fault management layers of all lanes.

The main task of the FML software is the fault management based on these communications. This task consists of error detection, error correction, lane isolation (in the case of an unrecoverable error), and lane reintegration.

Error detection is essentially based on a two round Byzantine distribution schema [9] where data are communicated between FMLs and voted using various specialized voters. The aim is to ensure that (1) all ASS instances of non-faulty lanes get identical messages from FML, (2) all AVI instances of non-faulty lanes get identical messages from FML, (3) for data calculated by all lanes (congruent source messages) all non-faulty lanes get the correct(ed) message, (4) for data calculated by one lane (single source messages) all non-faulty lanes get the correct(ed) message if the originator is not faulty. The implemented design allows detection of one Byzantine or deterministic fault in a four-lane system and recognition of a deterministic fault in a three-lane system.

2.2 FML Structure

The software architecture of the FML in one lane consists of a number of processes which communicate over a set of channels and jointly use a global memory and a separate global buffer. Figure 2 presents a simplified overview of this architecture. Each of the main processes itself is built from smaller subprocesses not shown in the figure, which communicate over local channels.

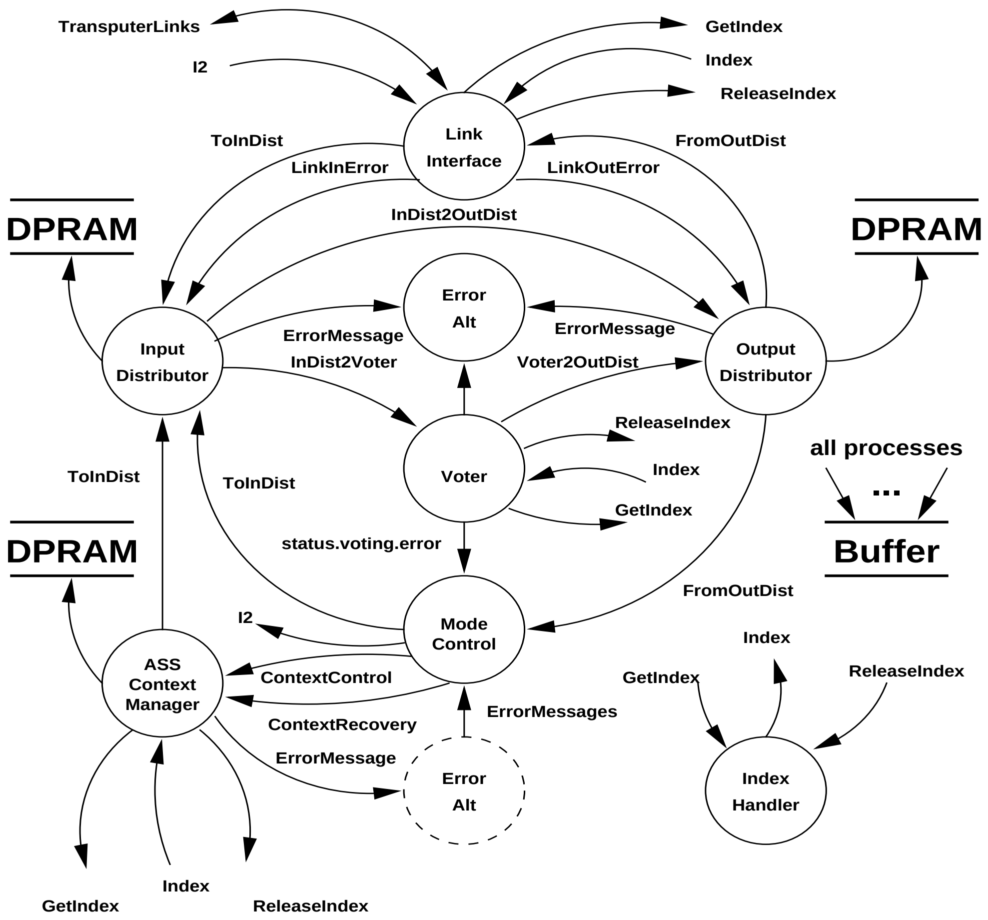

Figure 2. FML Processes

2.3 Deadlock Analysis

The overall goal of the analysis described in this article is to check whether none of the communicating system of processes that forms the FML software continuously blocks the communication on the vital channels. Precisely speaking, the following verification goal had to be investigated:

> In an environment that always accepts outputs from the system but may or may not refuse to provide inputs, the following assertion holds: Whenever the system reaches a stable state where all communications are blocked, all processes reading from vital channels are ready for input on these channels.

"Vital channels" are all those channels from Figure 2 which are still visible in Figure 3 below. Since the input transputer links used by the environment are vital, the verification goal implies that in a blocking situation the system is always ready for a new input from the environment, so this implies deadlock freedom of the system in the conventional sense.

3 Verification Approach

The specification language CSP (*Communicating Sequential Processes*) is associated with a formal method allowing to verify properties of systems of parallel processes. CSP is particularly suited for supporting the formal analysis of occam programs since it may be regarded as an abstract version of occam, associated with a formal semantics, allowing to give unambiguous interpretations of CSP specifications. The mathematical proof theory allows to verify properties of CSP specifications by means of logic reasoning. The CSP language, its mathematical foundations and its possible applications have been thoroughly investigated since the late sixties (see [7, 6, 5]). For the verification goal described above we apply five verification techniques that will be discussed in this section:

1. Model checking is used for the mechanized verification of small-size CSP components.
2. Abstraction is applied to "lift" occam process components to CSP components reflecting the essential aspects of the process communication behaviour while abstracting from details irrelevant for the verification goal.
3. The compositional proof theory of CSP is applied to derive global properties of the complete FML system from the local properties established for the isolated components.
4. Generic theories are applied to reduce the amount of proof obligations by showing that all process instances of a certain class inherit certain properties.
5. Algebraic reasoning is applied to transform CSP process specifications into equivalent ones better suited for the model checker.

3.1 Model-Checking with FDR

FDR (*Failures-Divergence Refinement*) is a model-checking tool for state machines, with foundations in the theory of concurrency based upon CSP. Except for the ability to check determinism [11], primarily for checking security properties, its method of establishing whether a property holds is to test for the refinement (in one of the semantic models of CSP) of a transition system capturing the required specification by the candidate machine. The main ideas behind FDR are presented in [10].

The notion of refinement is a particularly useful concept in many forms of engineering activity. If we can establish a relation between components of a system which captures the fact that one satisfies at least the same conditions as another, then we may replace a worse component by a better one without degrading the properties of the system. Refinement relations can be defined for systems described in CSP in several ways, depending on the semantic model of the language which is used. The relation to be used for goals as described here is *Failures Refinement* $P \sqsubseteq_F Q$, which is defined by the requirement that the failures (i. e., pairs of traces and associated refusal sets) of a refining process should be included in those of the refined process.

The FDR tool is well-established for verification purposes similar to the one described here, see [1].

3.2 Abstraction

For the analysis of deadlock freedom it is unnecessary to inspect every detail of the occam code, since only a subset of the statements of this code influences the communication behaviour. It is therefore possible to generate a CSP specification which represents an abstracted version of the original occam process P and shows only that amount of detail which is relevant for its communication behaviour. We call such a CSP specification $\mathcal{A}(P)$ a *valid abstraction* of the corresponding occam process P, if

> *Whenever P runs into a deadlock situation this implies that $\mathcal{A}(P)$ may run into a deadlock situation, too.*

If a valid abstraction $\mathcal{A}(P)$ is available, we only need to analyse $\mathcal{A}(P)$ instead of P: If $\mathcal{A}(P)$ is free of deadlocks, the same must hold for P.

The basic approach to construct valid abstractions uses four techniques in the translation from occam to CSP:

1. Every sequential algorithm whose results do not influence communication behaviour is deleted.
2. Each occam channel protocol is reduced to the set of values influencing the communication behaviour in a distinctive way.
3. Every occam IF-construct `IF condition THEN P ELSE Q` may be replaced by the equivalent if-construct in CSP or by the internal choice operator of CSP yielding $P \sqcap Q$.
4. If valid abstractions $\mathcal{A}(P)$, $\mathcal{A}(Q)$ for two processes P and Q are available and these interpretations use the same abstractions on their communication interface I, then $\mathcal{A}(P) \underset{I}{\parallel} \mathcal{A}(Q)$ is a valid abstract interpretation of P and Q

operating in parallel. Using this technique, larger abstractions can be build from existing ones.

The following table gives an overview of the translation between occam and CSP constructs.

occam-**construct**	**CSP-construct**
IF_THEN_ELSE	$\sqcap$ or if_then_else
PAR (PRIPAR)	$\vert\vert\vert$ or $\vert\vert$
ALT (PRIALT)	[]
WHILE	$P = \cdots \rightarrow P$ or $P =$ if b then $\cdots$ else SKIP
c?x	c or c?x
c!a	c or c!a
SEQ	; or $\rightarrow$

To show the applications of these techniques, we give an example including two parallel processes P and Q that communicate through the cannels a, b, and c, and have a common variable mc initialized to 0. At first, we present the occam definitions of P and Q.

P:

```
WHILE TRUE
  SEQ
    IF
      mc = 1
        a ! TRUE
      TRUE
        b ! FALSE
    c ? mc
```

Q:

```
WHILE TRUE
  SEQ
    IF
      mc = 1
        a ? x
        mc := 0
      TRUE
        b ? x
        mc := 1
    c ! mc
```

In the first abstraction, the occam IF-construct is simply converted to CSP internal choice $\sqcap$. Unfortunately, $S1$ is obviously not deadlock free, which can also be proved with FDR.

Abstraction 1:

$$
\begin{aligned}
P &= a \rightarrow c \rightarrow P \sqcap b \rightarrow c \rightarrow P \\
Q &= a \rightarrow c \rightarrow Q \sqcap b \rightarrow c \rightarrow Q \\
S1 &= P \underset{\{c,a,b\}}{\|} Q
\end{aligned}
$$

Since $S1$ is a valid abstraction of the occam process, two cases are possible: Either the occam process contains a deadlock or we have introduced too high a degree of nondeterminism in the construction of $S1$. Since we intuitively assume that the occam process should turn out to be deadlock free, we try to find another valid abstraction which is more deterministic than $S1$.

In the following abstraction, a new CSP process MC is introduced, which simulates the common variable mc of the occam process. Usually, two operations, *read* and *write*, can be applied to a variable. The channels *rp_mc*, *rq_mc* and *w_mc* implement these operations on the variable *mc*. The occam IF-construct is now interpreted as a CSP if-construct. It can be proved that $S2$ is deadlock free, which implies that the occam process ist deadlock free as well.

Abstraction 2:

$$
\begin{array}{ll}
MC & = MC1(0) \\
MC1(x) & = rp\_mc!x \rightarrow MC1(x) \\
 & \;[]\; rq\_mc!x \rightarrow MC1(x) \\
 & \;[]\; wq\_mc?y \rightarrow MC1(y) \\
 & \\
P & = rp\_mc?mc \rightarrow \\
 & \quad \text{if } (mc == 1) \text{ then } a \rightarrow c \rightarrow P \\
 & \quad \text{else } b \rightarrow c \rightarrow P \\
Q & = rq\_mc?mc \rightarrow \\
 & \quad \text{if } (mc == 1) \text{ then } a \rightarrow wq\_mc!0 \rightarrow c \rightarrow Q \\
 & \quad \text{else } b \rightarrow wq\_mc!1 \rightarrow c \rightarrow Q \\
S2 & = (P \underset{\{c,a,b\}}{\|} Q) \underset{\{|rp\_mc, rq\_mc, wq\_mc|\}}{\|} MC
\end{array}
$$

During generation of a CSP specification of an occam process, two important aspects have to be considered. On the one hand, the generated CSP specification should be a valid abstraction of the occam process; on the other hand, as many details which do not influence the communication behaviour of the occam process as possible should be omitted, in order to reduce the size of the state space of the CSP specification far enough to allow model-checking with FDR. This task requires considerable expertise from the person developing the abstraction, but large portions of the work can be automated using the translation heuristics described above.

3.3 Compositionality

CSP provides several high-level operators, such as parallel operator ($\|$) and interleaving operator ($|||$), that allow to construct new processes from existing ones. A crucial property of CSP is the fact that refinement is preserved under compositions involving these operators:

If $Q_i \sqsubseteq_F P_i$ for $i : 0..n$ and ω is an n-ary operator, then $\omega(Q_0, \ldots, Q_n) \sqsubseteq_F \omega(P_0, \ldots, P_n)$ holds.

This fact will be exploited for our analysis to partition the system state spaces that are too large to be represented by a single transition graph for model checking: If deadlock freedom of $\omega(P_0, \ldots, P_n)$ cannot be established because of the problem size, find simpler processes $Q_0, \ldots, Q_n$ such that $Q_i \sqsubseteq_F Q_i$ and $\omega(Q_0, \ldots, Q_n)$ is deadlock free. If this property can be established, compositionality combined with refinement implies the deadlock freedom of $\omega(P_0, \ldots, P_n)$ as well.

For our purpose, the compositionality of the parallel or interleaving operator is applied at two levels.

In order to proof that the overall system consisting of 8 main processes running in parallel is deadlock free we construct for each process P_i a valid abstraction $\mathcal{A}(P_i)$ and show that the parallel composition of these is deadlock free. Since even the system consisting of the abstractions is too large to be dealt with in the FDR model checker we find simpler processes Q_i such that each $\mathcal{A}(P_i)$ is a refinement of the corresponding Q_i. The Q_i are checked for deadlock individually; since they are deadlock free the refinement property ensures that the $\mathcal{A}(P_i)$ are deadlock free (internally), too. The essential proof now has to show that the parallel composition of the Q_i is deadlock free, which via the compositionality argument yields deadlock freedom of the system of $\mathcal{A}(P_i)$, and via abstraction and compositionality finally ensures the desired property of the original system.

In the next section we will introduce the Q_is and prove that the parallel composition

$$(Q_1 || \ldots || Q_8)$$

of the simplified processes is deadlock free.

3.4 Generic Theories

To increase the efficiency of the verification process, we use generic theories stating properties of process classes that are inherited by each instance of the class. Concrete occam processes can be identified as refinements of certain class instances. Since the properties established in the generic theories are preserved under refinement, we can conclude that the occam process inherits the desired property as well.

To give an example, the following theorem states that a system which is deadlock free will still have this property if *concentrators or multiplexers* are added to its output channels, provided that the following conditions hold:

- Each concentrator or multiplexer operates in strict alternation between the following two phases:
  1. Output phase: produce a number of outputs on a (sub-)set of pre-defined output channels. This number may be zero, but must be limited by a constant max. Then start the input phase.
  2. Input phase: wait for an arbitrary input without blocking any input channel. After having received exactly one input, go back to the output phase.
- The environment never blocks an output channel of the concentrator or multiplexer.

Theorem 3.1. *Let MUX be a concentrator/multiplexer process defined as*

$$MUX =_{df} MUXOUT(max, IN, OUT)$$

$$\begin{aligned} MUXOUT(n, IN, OUT) =_{df}\ & \mathbf{if}\ (n = 0) \\ & \mathbf{then}\ MUXIN(IN, OUT) \\ & \mathbf{else}\ (MUXIN(IN, OUT) \\ & \qquad \sqcap \\ & \qquad (\ \sqcap_{y:OUT}\ y \\ & \qquad \to\ MUXOUT(n-1, IN, OUT))) \end{aligned}$$

$$MUXIN(IN, OUT) =_{df} \Box_{x:IN}\ x\ \to\ MUX$$

for a fixed max. Let X be a process which is deadlock free in the context $RUN(\Sigma)$[1]
and has a (sub-)set of outputs into IN. Let CON be the context

$$CON =_{df} RUN(OUT)\ |||\ CHAOS(\Sigma - (\alpha(X) \cup IN \cup OUT))$$

Then

1. $(RUN(IN)\ |||\ CHAOS(\Sigma - IN))\ \sqsubseteq_F\ (MUX \underset{OUT}{\|} CON) \setminus OUT$ *holds.*
2. $(X \underset{IN}{\|} MUX) \underset{OUT}{\|} CON$ *is also deadlock free.*

This theorem is generic in the interfaces IN, OUT and in the maximum value max of consecutive outputs. It turns out that several of the processes shown in Figure 2 are refinements of the above multiplexer/concentrator process instantiated with different interfaces and maximum numbers.

3.5 Algebraic Reasoning

CSP possesses a proof theory which provides algebraic laws for the transformation of CSP processes into equivalent ones. Specifically, deadlock freedom is preserved under such term rewriting laws. Algebraic manipulations are useful to simplify process terms and reduce the state spaces of processes before model checking. Especially, after abstracting an occam process into a CSP specification, there are often situations in which several different occam cases are collapsed into one CSP case.

4 Verification Results: Deadlock Freedom of the FML

In this section we will present some results to prove that the FML of each isolated lane fulfills the verification goal stated in Section 2. The main result is stated in the following theorem.

[1] Σ denotes the set of all communications possible in the system, i.e. the union of all alphabets of subprocesses

Theorem 4.1. *In every context which never blocks outputs on transputer links, outputs to the ASS layer and outputs to the* AVI *layer, the following property holds:*

Whenever the FML *system reaches a stable state where no communications are possible, all processes reading from vital channels are ready for input on these channels.*

At first we will present two lemmas providing useful simplifications by structuring the FML of a lane into a number of sub-systems $S_1, S_2, \ldots, S_n$, such that the following conditions hold:

- S_{i+1} is a sub-system of S_i in the sense that there exist processes P_{i+1} such that $(S_{i+1} \ \| \ P_{i+1}) = S_i$.
- Let I_{i+1} be the interface between S_{i+1} and P_{i+1}. Then, if S_{i+1} is deadlock free in context CON_{i+1}, S_i will also be deadlock free in context CON_i.

Lemma 4.2. *Let S_1 be the* FML *sub-system of a lane consisting of all* FML *processes but* Index Handler *that is* FML $= (S_1 \ \| \ \text{Index Handler})$. *Define interface and context*

$$I_1 =_{df} \{|GetIndex[k], Index[k], ReleaseIndex[\ell] \mid k : 0..16, \ell : 0..7|\}$$

$$CON_1 =_{df} RUN(I_1)$$

Then, if $(S_1 \underset{I_1}{\|} CON_1)$ is free of deadlocks and no reader process permanently blocks its input channels, the same holds for the full FML *of one lane.*

According to this lemma, we need not consider the process Index Handler in proving the deadlock freedom of the whole system. Another lemma providing a useful simplification for the proof of Theorem 4.1 shows the possibility to delete the process Error ALT from the system without lossing the correctness of the theorem:

Lemma 4.3. *Let S_2 be the sub-system of S_1 consisting of all S_1-processes but* Error ALT *that is $S_1 = (S_2 \ \| \ \text{Error ALT})$. Define interfaces and context*

$$I_2 =_{df} I_1 \cup I_{21} \cup I_{22}$$
$$I_{21} =_{df} \{|ErrorMessages[k]|k : 0..6|\}$$
$$I_{22} =_{df} \{|AllErrorMessages|\}$$

$$CON_2 =_{df} (RUN(I_1 \cup I_{21}) \ ||| \ CHAOS(I_{22}))$$

Then, if $(S_2 \underset{I_2}{\|} CON_2)$ is free of deadlocks and no reader process permanently blocks its input channels, the same holds for $(S_1 \underset{I_1}{\|} CON_1)$.

The proofs of these lemmas follow from generic theories similar to the one described in the previous section.

There are two types of FML process behaviours regarding channel input:

– **Unconditional Input:** The process will always accept a new input on the channel, possibly after a bounded number of non-blocking internal communications. The acceptance of a new input does *not* depend on successful outputs to other channels prior to the next input. Unconditional input is provided whenever the corresponding process abstraction behaves as RUN on the input channel.
– **Bounded Conditional Input:** *Provided that the environment will not block process outputs*, the process will always accept a new input on the channel, possibly after a bounded number of non-blocking internal and external communications. This type of input behaviour is guaranteed whenever the corresponding abstraction behaves as $MUXOUT(max, IN, OUT)$ on its input channels IN and output channels OUT.

The occam processes Input Distributor and Output Distributor have only unconditional inputs. A valid abstraction of the occam process Input Distributor for example is a failures refinement of the following process:

$$(RUN(I_1) ||| CHAOS(I_2))$$

where

$$I_1 =_{df} \{| ToInDist[i], LinkInError[j] \mid i : 0..6, j = 0..4 |\}$$
$$I_2 =_{df} \{| InDist2Voter[i], InDist2OutDist \mid i : 0..3 |\}$$

All channels in I_1 are unconditional.

All other occam processes are failures refinements of some multiplexer/concentrator. ASS Context Manager, for instance, refines the following process:

$$MUXOUT(10, \{Context.recovery, Context.control\}, \{ToInDist[6]\}).$$

Context.recovery and *Context.control* are bounded conditional inputs. The similar processes are Voter, Link Interface, ASS Interface and Mode Control.

Proof sketch of Theorem 4.1.
The two preceding lemmas can be applied to conclude that the remaining channels and processes that are still to be analysed with respect to deadlock freedom are the ones depicted in Figure 3, where unconditional input is depicted by a dashed arrow and bounded conditional input is marked by a solid arrow, labelled by the channel name.

Analysis of the data flow diagram and the process behaviours regarding channel input shows that every cycle in the diagram contains at least one channel where the corresponding reader process provides unconditional input. Furthermore, every process provides at least bounded conditional input. As a consequence, a blocking situation on a cycle can never occur, as long as a process is still willing to output. Therefore a stable state without active communications implies that all the processes shown in Figure 3 are waiting for new inputs. These input channels include the set on vital channels. This completes the proof sketch of Theorem 4.1 (detailed proofs can be found in [3]).
□

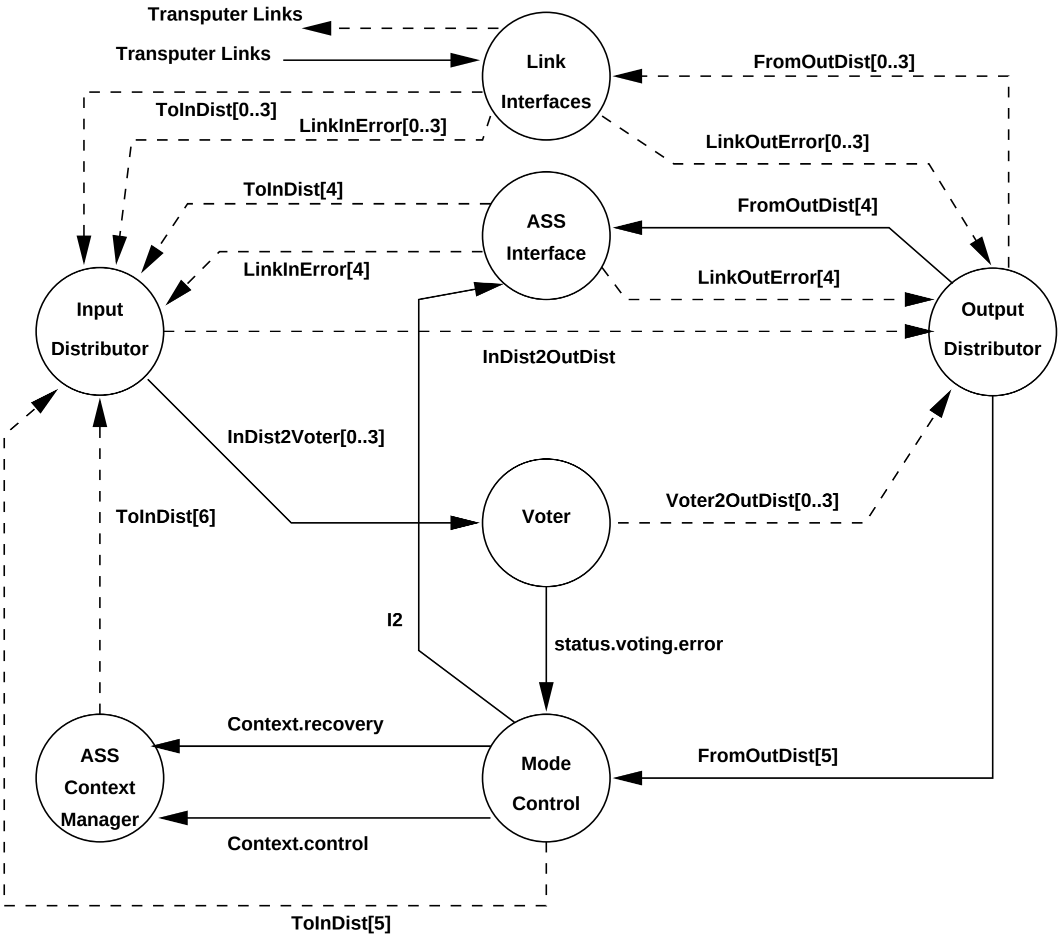

Figure 3. Data flow graph for deadlock analysis in Theorem 4.1

5 Conclusion

5.1 Experiences

The overall experience with this use of CSP specification and model-checking for the verification of realistic occam software is very positive. Despite the size and complexity of the object to be analyzed it was possible to use the approach by applying abstract interpretation and a suitable modularization for splitting the task into manageable subtasks. The result is a low cost/high quality verification of an important aspect of a real-world project. The following points justify this claim:

Real-world project: the collection of all relevant parts of the code accumulates to about 11,600 lines of occam code for FML. For the AVI layer

which has about the same size, a similar verification was performed.
Low cost: The effort for achieving the same verification results by means of conventional reviewing and testing techniques would have been at least four times the effort invested by the team applying our approach. At the same time we doubt whether the degree of trustworthiness achieved by the method described here would be reached at all using conventional techniques.
High quality: The verification of AVI and FML uncovered a number of deadlock situations that had neither been found nor could be retraced by testing.

5.2 Future Work

The main emphasis on our future work related to verification projects similar to the one described here is put on the following goals:

- Improve the degree of automation for the abstraction process occam to CSP.
- Improve the tool support for the proof of generic theories. This will be performed in the context of the UniForM project [13], based on the HOL/Isabelle tools [8, 12].
- Develop efficient verification suites using series of verification and test subgoals to cope with the complexity of the full verification task.

Acknowledgements. We would like to thank Rachel Cardell-Oliver for her extremely helpful suggestions related to the use of generic theories. Furthermore, we would like to thank the members of the UniForM project for providing a framework for the scientific aspects of this work. Discussions with the colleagues in the project helped to focus on essential points. Last but not least, we are grateful for the support of the development team at DASA Bremen, who were always ready to clarify details of the software and the overall design of FML. This work has been partially supported by the German Ministry of Education and Research (BMBF) as part of the project UniForM under grant No. FKZ 01 IS 521 B2.

References

1. N. A. Brock and D. M. Jackson: Formal Verification of a Fault Tolerant Computer. In *Proceedings of 1992 Digital Avionics Systems Conference.* (1992)
2. B. Buth and J. Peleska: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Verification of Avionics Interface AVI.* Technical Report, JP Software-Consulting, (1996).
3. B. Buth, J. Peleska and H. Shi: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Fault Management Layer (FML): Verification of Deadlock Freedom.* Technical Report, JP Software-Consulting, (1996).
4. Daimler-Benz Aerospace: *DMS-R FTC Detailed Design Document Volume 3 (FML Software)*
5. J. Davies: *Specification and Proof in Real-Time CSP.* Cambridge University Press (1993).

6. Formal Systemes: *Failures Divergence Refinement FDR2* Preliminary Manual. Formal Systems (Europe) Lts (1995).
7. C. A. R. Hoare: *Communicating Sequential Processes.* Prentice-Hall Internationaal (1985).
8. Kolyang, C. Lüth and B. Wolff: Generic Interfaces for Formal Development Support Tools. To appear in *Proceedings of the International Workshop for Tool Support in Verification and Validation.* LNCS (1997).
9. L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*, In: ACM Transactions on Programming Languages and Systems, Vol.4, Nr. 3, (1982)
10. A. W. Roscoe: *Model-Checking CSP.* In: A Classical Mind, Eassys in Honour of C.A.R. Hoare. Prentice-Hall Internationaal (1994).
11. A. W. Roscoe: *CSP and determinism in security modelling.* In: IEEE Symposium of Security and Privacy, (1995).
12. H. Tej and B. Wolff: A Corrected Failures-Divergence Model for CSP in Isabelle/HOL. To appear in *Proceedings of the Formal Methods Europe*, LNCS (1997).
13. B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer and A. Baer. Universal Formal Methods Workbench. In U. Grote and G. Wolf, editors, *Statusseminar Softwaretechnologie des BMBF, March 1996, Berlin*, Deutsche Forschungsanstanlt für Luft- und Raumfahrt, Berlin, 1996.