Combining Methods for the Livelock Analysis of a Fault-Tolerant System

Bettina Buth*, Jan Peleska**, and Hui Shi***

Abstract. This article presents experiences gained from the verification of communication properties of a large-scale real-world embedded system by means of formal methods. This industrial verification project was performed for a fault-tolerant system designed and implemented by Daimler-Benz Aerospace for the International Space Station ISS and focussd essentially on deadlock and livelock analysis. The approach is based on CSP specifications and the model-checking tool FDR. The tasks are split into manageable subtasks by applying abstraction techniques 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

The acceptance of Formal Methods in industries essentially depends on their scalability, i.e. their applicability in large scale realistic industrial projects. An important aspect is the availability of suitable tools, but from our experience this is but one aspect. The diverse nature of system components and the techniques used in the different steps of the development process require the use of a combination of methods for the development as well as for the analysis of these components. In this paper we report experiences in using a combination of methods for the analysis of a large software system, namely the fault-tolerant data management system for the International Space Station (ISS). It was necessary to use these methods in order to break down the respective tasks into manageable sub-tasks.

The project started in 1995 when 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 ISS. Up to now various aspects of the system correctness have been investigated:

- freedom of deadlock (see e.g. Buth et al. [3]),
- freedom of livelock,
- correct implementation of voting algorithms,
- correct implementation of the Byzantine Agreement Protocol,

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

– perfomance properties depending on clock rates,
– Hardware-In-The-Loop tests for the overall system.

Different approaches were employed for these analysis tasks. These comprise abstraction of the occam code to CSP processes and reasoning on these abstractions, Hoare-style program verification for sequential code, and the use of stochastic Petri nets for the throughput analysis.

In contrast to the deadlock analysis presented in Buth et.al. [3], this article focusses on the techniques used for the livelock analysis. In the following section a brief overview of the fault tolerant computer is given. Section 3 provides an overview of the methods used for the analysis. In Section 4 our experiences are described and put into relation to the deadlock analysis. The conclusion outlines future work, especially with respect to tool support for similar analysis projects.

2 The Fault Management System

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.

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.

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.

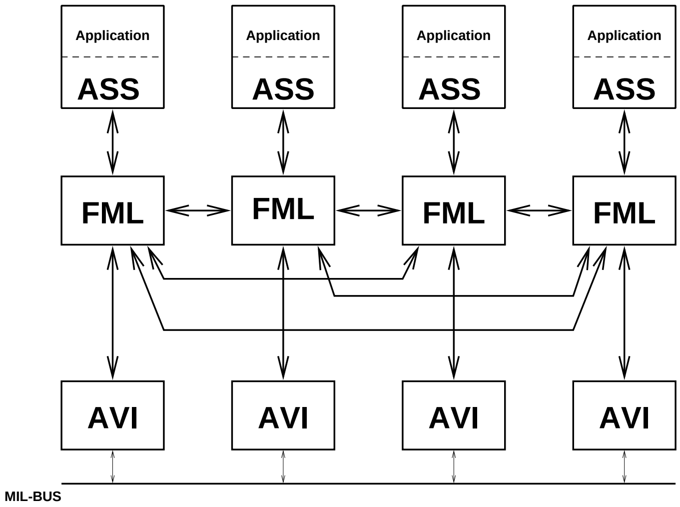

Figure 1. FTC architecure

3 Livelock Analysis With CSP

In high level concurrent programming languages, such as occam it is conventional for communication channels between two processes to be concealed from the environment. This can potentially cause a form of divergence known as *livelock*[1]

The objective of our analysis is to investigate occurrence of internal divergence only. A CSP (or occam) system X is called *livelock free* with respect to interface channels $c_1, c_2, \ldots, c_n$, if the system will never engage in an unbounded sequence of (internal) communications without interleaved (visible) communications on the interface $c_1, c_2, \ldots, c_n$. This property is as important as the *absence of deadlocks* since it ensure the interaction of the system with its environment.

For real-time applications, periodic internal events such as time-scheduled interrupts cannot cause divergence as long as their period is long enough to allow for other activities in between. These events may cause cycles of internal communications which are intended and not to be considered as livelock, because their occurrence is guarded by time intervals where other communication events are possible. To reflect this properly in an untimed CSP model, we regard these channels as part of the interface for the purpose of livelock analysis.

The verification of livelock freedom of AVI and FML is discussed in this document. We wish to justify that they are livelock free with respect to their

[1] TECHNICAL NOTE: Observe that the CSP semantics of the failures-divergence model defines divergence using the "chaotic" process $\perp$ which may show the behaviour of *internal divergence* (no visible events at the interface but unbounded internal communication sequences) and *external divergence* (arbitrary events may occur or may be refused at the interface).

respective external interfaces. In the following we are going to present the techniques applied for this purpose. The first subsections describe methods used for both AVI and FML, whereas Subsection 3.5 and Subsection 3.6 refer to different techniques used in FML and AVI respectively for deriving results for larger sets of processes that can not be obtained using model-checking.

3.1 Abstract Interpretation

For the analysis of livelock freedom it is unnecessary to inspect every detail of the occam code, since only a subset of the programmed statements influences the communication behaviour. It is therefore possible to generate a CSP specification which represents an abstracted version of the original occam process P showing only the amount of detail which is relevant for communication behaviour. Such a CSP specification $\mathcal{A}(P)$ is called a *valid abstract interpretation* of the corresponding occam process P, if

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

If a valid abstract interpretation $\mathcal{A}(P)$ is available, we can analyse $\mathcal{A}(P)$ instead of P: If $\mathcal{A}(P)$ is free of livelocks, the same must hold for P.

The basic approach to construct valid abstract interpretations 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 different way.
3. Every occam IF-construct `IF condition THEN P ELSE Q` may be replaced by the internal choice operator of CSP yielding $P \sqcap Q$.
4. If valid abstract interpretations $\mathcal{A}(P), \mathcal{A}(Q)$ 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 fact, larger abstract interpretations can be built from existing ones.

If a valic abstraction of a occam process is not livelock free, two cases are possible: Either the occam process contains a livelock or we have introduced too high a degree of nondeterminism when applying the first three techniques in the construction of its CSP abstraction. Since we intuitively assume that the occam process should turn out to be livelock free, we try to find another valid abstraction which is more determnistic than the one we have constructed. A more detailed description of our abstraction methods is given in [3].

3.2 Model-Checking and Refinement Properties

Every CSP specification consisting of finite-state processes with finite-value channels can be translated into a finite *transition graph* representation. This graph contains all the semantic information of the original CSP specification. As a consequence, every property of the specification – as, for example, livelock freedom – can be verified by exhaustive analysis of the transition graph. Moreover, such an analysis can be mechanised. The FDR tool provides this mechanisation and has been used for all model checking results about the abstract interpretations described in this document.

Two CSP processes P_1 and P_2 can be compared, whether

> *P_2 can perform only communications P_1 may also perform, but P_2 refuses less and diverges less than P_1.*

If this is the case, P_2 is called a *failures-divergence refinement* of P_1. This is denoted by $P_1 \sqsubseteq_{FD} P_2$. Failures-divergence refinement preserves deadlock freedom and livelock freedom and restrictions on possible communication sequences: If P_1 is free of livelocks and observes a safety condition about communication sequences, then every process refining P_1 in the failures-divergence model has these properties, too. If P_1 and P_2 are finite-state CSP specifications the refinement relation $P_1 \sqsubseteq_{FD} P_2$ can be verified by model checking.

A typical application of refinement in our context is the fact that

> *given a valid abstract interpretation $\mathcal{A}(P)$ of P and a process Q such that $Q \sqsubseteq_{FD} \mathcal{A}(P)$, then Q is also a valid abstract interpretation of P.*

A second version of refinement applied to prove livelock freedom is *trace refinement*. Process P_2 is a trace refinement of P_1 (denoted by $P_1 \sqsubseteq_T P_2$), if every trace of communications that P_2 may perform is also a trace of P_1. Failures-divergence refinement implies trace refinement, but for certain situations it is easier to deal with the latter, because the investigation of trace refinement properties does not require the analysis of refusal sets which is necessary for failures-divergence refinement checking. Note that formally speaking, livelock freedom means that there exists a *bounding function* $f : \mathbb{N}_0 \to \mathbb{N}_0$, such that

$$\#(s \upharpoonright L) \leq f(\#(s \upharpoonright \{\!|c_1, c_2, \ldots, c_n|\!\}))$$

where $\#(s \upharpoonright L)$ is the number of local communications L in s and $\#(s \upharpoonright \{\!|c_1, c_2, \ldots, c_n|\!\})$ denotes the length of the trace s restricted to communications on channels c_i. An equivalent way of expressing the property of livelock freedom is to say that $X \backslash L$ must not diverge.

Our main application of trace refinement thus is given by the following theorem which obviously holds because the refining process only runs through traces which are possible for the abstract process as well:

Theorem 3.1. *Let P_1 be a process over alphabet $\{|c_1, c_2, \ldots, c_n|\} \cup L$ with bounding function $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ such that every trace s of P_1 satisfies*

$$\#(s \upharpoonright L) \leq f(\#(s \upharpoonright \{|c_1, c_2, \ldots, c_n|\}))$$

Then f is also a bounding function for every trace of P_2 which refines process P_1.
□

As a consequence of this theorem, we can apply the following verification strategy to prove that a process P_2 is free of livelocks:

- Construct a "simple" process P_1 for which the existence of a bounding function is obvious.
- Verify by model checking that $P_1 \sqsubseteq_T P_2$ holds. This implies livelock freedom of P_2.

In general it is helpful to take simple processes from generic theories with known properties as the goal of this further abstraction. Examples for such generic theories are pipes and buffers, or the more specialized processes $MUXOUT$ below.

In the following we are going to take the process *Input Distributor*, which is one of the modules of FML, as an example to show the application of Theorem 3.1. The structure of *Input Distributor* is given by Figure 2.

At first we give a general CSP specification

$$MUXOUT_{tr}(max, max, IN, OUT)$$

of processes which will always be ready for inputs on channels IN after at most max outputs on channels OUT.

$$\begin{aligned} MUXOUT_{tr} \quad & (n, max, IN, OUT) = \\ & if\ (n == 0) \\ & then\ MUXIN_{tr}(max, IN, OUT) \\ & else\ (MUXIN_{tr}(max, IN, OUT) \\ & \qquad [\,]\ ([\,]y : OUT @ (y \rightarrow MUXOUT_{tr}(n-1, max, IN, OUT)))) \end{aligned}$$

$$\begin{aligned} MUXIN_{tr}(max,\ & IN, OUT) = \\ & [\,]x : IN @ (x \rightarrow MUXOUT_{tr}(max, max, IN, OUT)) \end{aligned}$$

Let OUT be L and IN be $\{|c_1, \cdots, c_n|\}$, then a bounding function f of the CSP specification $MUXOUT_{tr}(max, max, IN, OUT)$ is $f(x) = max * x$. Now we can use the FDR tool for verifying the following refinement relation:

$$MUXOUT_{tr}(4, 4, IN, OUT) ||| CHAOS(M) \sqsubseteq_T \ Input\ Distributor$$

where

$$\begin{aligned} IN \ &= \{|ToInDist|\} \\ OUT &= \{|To\_SS\_FIFO, To\_N\_FIFO, To\_QCS\_FIFO, To\_CS\_FIFO, \\ & \quad\ To\_OutDist\_FIFO|\} \\ M \ &= \{|Error\_Messages.0, LinkInError|\} \end{aligned}$$

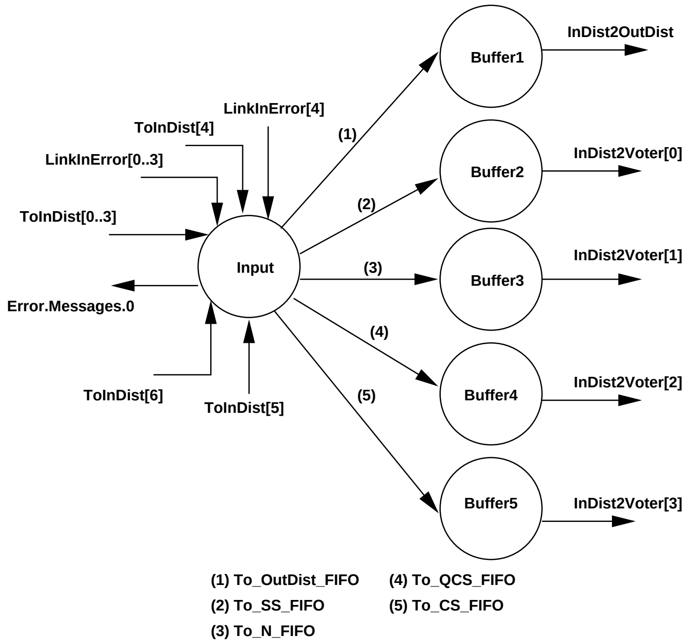

Figure 2. Input Distributor

and M is the set of channels whose communication behavior is irrelevant. So, the outputs on channels $To\_SS\_FIFO, To\_N\_FIFO, To\_QCS\_FIFO, To\_CS\_FIFO$ and $To\_OutDist\_FIFO$ are bounded by the inputs on channels $ToInDist$.

3.3 Compositionality and Generic Theories

A real-world system, such as FML, consists usually of a number of processes which communicate with each other, and each of them maybe further decomposed into several subprocesses, such as the process *Input Distributor*. How to compose the analysis results for each of these subprocesses and processes for the livelock verification of the overall system is the most difficult activity during the livelock analysis. In the following we will present some techniques und theoretical results we applied.

The first result follows from the simple theory of non-divergent pipes described in [10, p. 156]:

Theorem 3.2. *Suppose* CSP *process P is structured as*

$$P = (X \underset{LEFT}{\|} B \underset{RIGHT}{\|} Y)$$

where X is inputting from a set of channels IN and placing data on channels $LEFT$ into a buffer B and Y is inputting from B on $RIGHT$ and outputting on channels OUT without feedback to IN. Then, if

1. *the buffer B has bounded capacity,*
2. *the number of buffer inputs on $LEFT$ is bounded by the number of P-inputs on IN,*
3. *the buffer outputs on $RIGHT$ are destructive (that is, each entry is removed from the buffer by reading it on a $RIGHT$-channel),*

the number of B-outputs on $RIGHT$ is bounded by the number of X-inputs on IN. If in addition

4. *the number of Y-outputs on OUT is bounded by the number of entries in the buffer,*

the number of Y-outputs on OUT is also bounded by the X-inputs on IN.
□

Using the first of these results we can show that the outputs on channels $InDist2Voter$ and $InDist2OutDist$ in the process *Input Distributor* are bounded by the inputs on channels $ToInDist$, since all buffers involved are bounded buffers.

CSP provides several operators that allow to construct new processes out of existing ones. A crucial property of CSP is the fact that refinement is preserved under compositions involving these operators:

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

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

3.4 Algebraic Reasoning

CSP possesses a proof theory which provides algebraic laws for the transformation of CSP processes into equivalent ones. Specifically, livelock freedom is preserved under such term re-writing laws. Algebraic manipulations are useful to simplify process terms before model checking.

3.5 Liveness Induction

Another result which we applied during the FML analysis allows to derive livelock freedom of a collection of *triple-disjoint* communicating processes, where no event requires the participation of more than two processes, from livelock freedom of the isolated processes has been developed by Roscoe (detailed in [7]):

Theorem 3.3. *Suppose $V = \langle P_1, \ldots, P_n \rangle$ is a triple-disjoint system of non-divergent processes such that for every P_i in V*

$$P_i \backslash (\bigcup_{j<i} (\alpha(P_i) \cap \alpha(P_j)))$$

is divergence-free. Then

$$(P_1 \;\|\; \ldots \;\|\; P_n) \backslash H \text{ with } H = (\bigcup_{j<i} (\alpha(P_i) \cap \alpha(P_j)))$$

is also divergence-free.
□

Informally speaking, this theorem states that if no process in the system can perform an infinite sequence of communications with its predecessors then the system is livelock free with respect to its incoming and outgoing channels.

Theorem 3.3 is found to be useful in many cases, but to find an order of the processes of a parallel system may be very ineffective. A FML lane consists of 8 processes which have a complicated communication behavior. It is very difficult to arrange them in a proper order for applying the theorem. One should try a large number of combinations (at worst 8!) and there are 7 proof obligations for each combination.

For the reason above we have developed a new technique called *liveness induction*, which can be used to verifiy livelock freedom inductively at the channel level. To verify that process P is livelock free with respect to channels of interface I, the induction is based on a set $\mathcal{L}$ of channels, inductively defined by

1. The interface I is contained in $\mathcal{L}$.
2. If $\{\!|c_1, \ldots, c_n|\!\}$ is contained in $\mathcal{L}$ and there exist a channel $d \notin I$ and a bounding function $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$, such that
$\#(s \upharpoonright \{\!|d|\!\}) \leq f(\#(s \upharpoonright \{\!|c_1, \ldots, c_n|\!\}))$ then d is contained in $\mathcal{L}$, too.

Liveness induction is now applied according to the rules of the following theorem (see Peleska et.al. [13]):

Theorem 3.4. *Let $P_1, \ldots, P_n$ be the set of processes of a system of concurrent processes, such that each P_i is livelock free with respect to its local interface $I_i =_{df} \alpha(P_i) - L_i$, where L_i is the set of its internal channels. Then $(P_1 \;\|\; \ldots \;\|\; P_n)$ is non-divergent with respect to global interface $I \subseteq \bigcup_{i=1}^{n} I_i$, if $\mathcal{L} =_{df} \bigcup_{i=1}^{n} I_i$ can be inductively constructed according to the rules given above.*
□

After showing the livelock freedom of each top-level process of FML we treat all the processes except the process $Link\ Interface$ as a single process, say $REST$. Let $SYS = \langle REST, Link\ Interface \rangle$, it is a triple-disjoint system. The following assertion can be proved using FDR:

$$Link\ Interface \backslash (\alpha(Link\ Interface) \cap \alpha(REST)) \text{ is free of livelock}$$

If we can show $REST$ is livelock free , then the whole system is livelock free, too, using Theorem 3.3.

The next verification target is to show the livelock freedom of $REST$, which can be arrived by liveness induction. For instance, we know $ToInDist$ are external channels for $REST$, so they are in the Interface I of the process $REST$ and contained in $\mathcal{L}$. Since we have shown that communications on channels $InDist2Voter$ and $inDist2OutDist$ are bounded by the communications on $ToInDist$ in $Input\ Distributor$, so $InDist2Voter$ and $inDist2OutDist$ are contained in $\mathcal{L}$, too. At last we can show all channels are containted in $\mathcal{L}$.

Note that liveness induction is a forward analysis for livelock freedom, which is in contrast to the approach described in the next section.

3.6 Dependency Analysis

Whereas the software structure of FML allowed to use the repeated abstraction by using generic processes $\mathtt{MUXOUT}_{tr}$ and livelock induction, the complex communication behaviour of AVI required a different approach. It turned out that there are three obstacles:

- even some of the main processes are too complex to be verified using model-checking directly;
- it is not possible to decompose the task into subtasks based on the compositional theory since the communication behaviour turns out to be too complex;
- for the same reason no further abstraction is possible.

This combination prevents the applicability of liveness induction as presented in the previous section. As an alternative a special form of dependency analysis was used for investigating cycles in the communication graph of processes which could not be checked using FDR directly due to the size of the state space. This analysis is a backwards analysis that determines all possible chains of communication events that may lead to a specific communication. The analysis technique is derived from tree analysis techniques as the *Fault Tree Analysis* (for the use of Fault Tree Analysis in software applications see e.g. Lyu [12])

Consider a set of processes $P_1, \ldots, P_n$ which have been proved to be free of livelock individually. Let $SYS = P_1 \parallel \ldots \parallel P_n$ be a net of these processes with internal synchronisation communications $\mathcal{L}$. The *communication graph* of SYS displays the channels and the direction of communication between the processes $P_i, i \in \{1, \ldots, n\}$. If this graph for SYS contains cycles it is not necessarily the case that these cycles can actually occur. If this is the case for all cycles SYS is livelock free. For the analysis of AVI there are two reasons that a cycle can not actually occur:

- the CSP processes involved do not produce traces that correspond to the cycle; this is e.g. the case if one of the communications does not depend on its predecessor in the cycle,

– the cycle is possible in the CSP process, but additional information derived from the occam code allow to exclude the occurrence of this situation.

For the analysis of AVI most of the cycles could be invalidated due to the first cause. The approach is illustrated in the following using a small example from AVI.

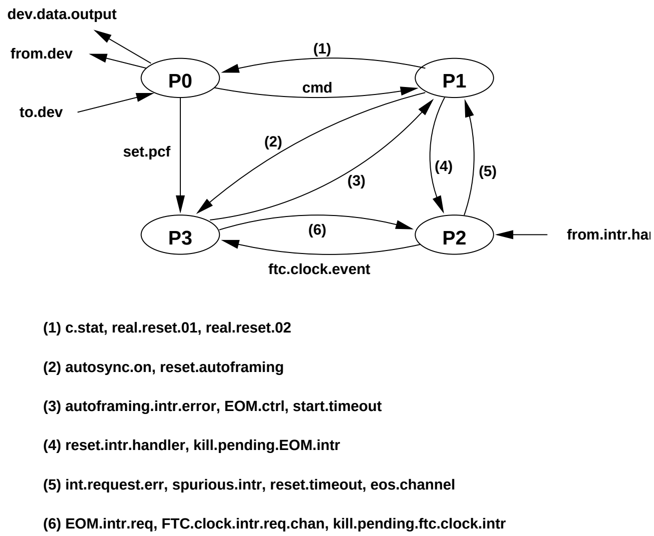

Figure 3. BC communication graph

The structure of module BC (broadcast unit) of AVI is presented in the communication graph of Figure 3. The individual processes `Pi`, $i \in \{0, \ldots, 3\}$ can easily be checked for livelock freedom using FDR, i.e. it is guranteed that no non-terminating internal communications can occur within the `Pi`. Each of these processes is itself a system of several subprocesses, which are denoted $P_{i1}, \ldots, P_{in_i}$. In order to ensure that the whole BC module is free of livelock it suffices to prove that none of the cycles in the graph can actually occur[2].

Regard e.g. the cycle involving `P0` and `P1`. A lifelock situation can only occur if output `cmd` would directly depend on the events `c_stat`, `real.reset.01`, or `real.reset.02` and vice versa. But this is not the case as code inspection of the CSP abstraction shows: the output of `cmd` only depends on the external input `to_dev`. In order to prove this it is necessary to trace the preceding communications leading to the output through the internal subprocesses of `P0`.

This type of analysis can be performed systematically for all cycles. In general it is only necessary to find one edge in a cycle that does not depend on its

[2] This does not guarantee a fair treatment of the inputs; such a property has to be proved using different means.

predecessors. This breaks a cycle and makes it uncritical with respect to livelock. The analysis itself is documented using techniques gleaned from fault-tree analysis. For each relevant communication it is traced by which other communications, resp. sequence of communications it can be triggered. E.g. the event `kill_pending_EOM_intr` from `P1` to `P2` has the following dependencies:

kill_pending_EOM_intr
- $\xleftarrow{\boxed{\text{P12}}}$ start_timeout.1
  - $\xleftarrow{\boxed{\text{P31}}}$ ftc_clock_event
    - $\xleftarrow{\boxed{\text{P22a}}}$ from_intr_handler $\boxed{\text{OK}}$
- $\xleftarrow{\boxed{\text{P12}}}$ start_timeout.2
  - $\xleftarrow{\boxed{\text{P33}}}$ (P33loop after enabled) start_subframe_timer
    - $\xleftarrow{\boxed{\text{P31}}}$ ftc_clock_event
      - $\xleftarrow{\boxed{\text{P22a}}}$ from_intr_handler $\boxed{\text{OK}}$

Note that it is necessary to trace the communications through several subprocesses not only of process `P1` and `P2` but also of process `P3`. The dependencies are noted as follows:

> For channels $ch_1, \ldots ch_n$ and process P
>
> $$ch_1 \xleftarrow{\boxed{\text{P}}} ch_2$$
>
> denotes that in P an output $ch1$ is produced if beforehand an input on channel $ch2$ did occur.
>
> ch_1
> $\xleftarrow{\boxed{\text{P}}}$ ch_2
> ...
> $\xleftarrow{\boxed{\text{P}}}$ ch_n
>
> means that an output on channel ch_1 is only possible after any of the inputs $ch_i, i \in \{2, \ldots, n\}$.

Note that it is essential that the dependency analysis is complete, i.e. that every sequence of communication leading to the output are investigated. This means that in the example above `start_timeout.1` and `start_timeout.2` are the inputs to process `P1` which can cause an output of `kill_pending_EOM_intr`.

In the case of `BC` and other units of the AVI it can be shown using this kind of analysis that most of the cycles in the communication graphs do not occur and it is possible to identify critical cycle situations as well. For AVI all the latter situations could be excluded due to further reasoning based on the code.

4 Livelock Analysis - Some Experiences

Starting point for the livelock analysis for both AVI and FML were the abstractions that had been used in the deadlock analysis (see Buth et.al. [4]). Due to the hierarchical architecture of both components the freedom of livelock for each of the main processes was tried to be established first. During this phase some adjustments in the abstractions were necessary to eliminate non-determinism and formalize the new proof obligations. One example is the explicit introduction of timer processes in order to avoid divergence through timer dependend events. These did not pose any problems for the the deadlock analysis.

During this first step it became soon obvious that for the subprocesses of both FML and AVI it was not feasible to use model-checking directly, even in cases where it was possible for the deadlock analysis. On the one hand this is due to the changes in the abstraction which enlarge the state space, on the other hand the problem arises since the states themselves are larger. The reason for this is that livelock analysis uses the *failure-divergence* model of CSP, while the deadlock analysis could be performed within the *failures* model. The internal representation of the states has to contain the additional information about the divergence sets and thus is larger. In some cases it was thus not possible to derive even the results for the main processes of FML and AVI. Additionally it was necessary to find a suitable approach of combining the results from the livelock analysis of the main processes to obtain a result for the two units as a whole.

In general two different approaches are possible in this situation:

- further abstraction and exploitation of the theorems of preservation of the results under refinement,
- further decomposition of components, separate analysis for each basic unit, and deriviation of properties for the combined units.

While the first proved to be a suitable way of dealing with the main processes of FML the complex communication behaviour of AVI made it necessary to pursue the second approach. In both cases it was necessary to employ suitable means for combining the results for the overall unit.

5 Conclusion

The main obstacle for the analysis of livelock freedom or other properities of realistic industrial systems is their size and complexity. Without suitable approaches for splitting the task into manageable subtasks the attempt of using formal methods and related tools in this area is futile. For our project is proved to be necessary to use a combination of techniques in order to decompose the goal appropriately. These techniques comprise

- abstraction of occam code to CSP,
- algebraic transformations,

– use of the compositional proof theory of CSP,
– abstraction to instances of processes from generic theories,
– model checking (for subprocesses small enough),
– liveness induction
– dependency analysis

While there is tool support for model checking, namely the system FDR, the other techniques had to be applied manually. Future work will comprise the search and if necessary development of tools supporting these tasks.

One useful tool would be on that interactively guides the abstraction process, not only for occam programs but for other compatible languages as well. In order to gain more confidence in the abstraction process, it will be necessary to theoretically justify the preservance of specification properties by the abstraction steps. The idea is to classify abstractions with respect to livelock, deadlock, and general safety properties. This will allow to reach a basis for the other methods in a systematic and even tool supported way.

Another aspect of abstraction is the use of generic theories. Here it will be necessary to look for more generic patterns and their specific properties. Some such patterns like buffer, systolic arrays, and multiplexer could be found in the software of our project, others will be found during other case studies. Again it will be useful to classify these theories with respect to their specific properties in order to develop heuristics for their application in context of abstractions. Furthermore, it will be necessary to prove the properties for the generic theories in a formal way. We would like to employ tools like HOL, Isabelle, or PVS for this task. Some work in this direction can e.g. be found in Buth et al. [2].

Tool support is also feasible for the dependency analysis. A rough idea is to use an approach with three phases:

– in the first phase the graph is build from the abstraction, i.e. the CSP processes;
– in the second phase all cycles are marked;
– in the third phase the cycles are investigated with respect to the dependencies of their constituent edges.

This kind of backwards cycle analysis could be complemented by a forward analysis for determining the bounds of the communication load of the cycles. This could help or replace the livelock induction process. Tools for these tasks might be found with groups working with graph transformation systems or in the area of graph theory in general.

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. Bettina Buth, Rachel Cardell-Oliver, and Jan Peleska. Combining tools for the verification of fault-tolerant systems. In Rudolf Berghammer Bettina Buth and Jan Peleska, editors, *Tools for Software Development and Verification*, volume 1 of *BISS Monographs*. Shaker-Verlag, 1996. (in print).

3. Bettina Buth, Michel Kouvaras, Jan Peleska, and Hui Shi. Deadlock analysis for a fault-tolerant system. In Michael Johnson, editor, *Algebraic Methodology and Software Technology. Proceedings of the AMAST'97*, number 1349 in LNCS, pages 60–75. Springer, December 1997.
4. B. Buth and J. Peleska: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Verification of Avionics Interface AVI.* Technical Report, JP Software-Consulting, (1996).
5. 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).
6. Daimler-Benz Aerospace: *DMS-R FTC Detailed Design Document Volume 3 (FML Software)*
7. N. Dathi: *Deadlock and Deaklock-Freedom,* Oxford University, D. Phil Thesis,1990.
8. J. Davies: *Specification and Proof in Real-Time CSP.* Cambridge University Press (1993).
9. Formal Systemes: *Failures Divergence Refinement FDR2* Preliminary Manual. Formal Systems (Europe) Lts (1995).
10. C. A. R. Hoare: *Communicating Sequential Processes.* Prentice-Hall Internationaal (1985).
11. L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*, In: ACM Transactions on Programming Languages and Systems, Vol.4, Nr. 3, (1982)
12. M. R. Lyu (ed.): *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, Computing McGraw-Hill
13. J. Peleska and H. Shi: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Fault Management Layer (FML): Verification of Livelock Freedom.* Technical Report, JP Software-Consulting, (1998).
14. A. W. Roscoe: *Model-Checking CSP.* In: A Classical Mind, Eassys in Honour of C.A.R. Hoare. Prentice-Hall Internationaal (1994).
15. A. W. Roscoe: *CSP and determinism in security modelling*. In: IEEE Symposium of Security and Privacy, (1995).