Combining Methods for the Analysis of a Fault-Tolerant System

Hui Shi
Bremen Institute of Safe Systems
University Bremen, Germany
shi@informatik.uni-bremen.de

Jan Peleska
Verified Systems International
GmbH, Bremen, Germany

Michel Kouvaras
DaimlerChrysler Aerospace RI
Bremen, Germany

Abstract

This paper presents experiences gained from the verification 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 DaimlerChrysler Aerospace for the International Space Station ISS. The verification involved various aspects of system correctness, like deadlock and livelock analysis, correct protocol implementation, etc. The approach is based on CSP *specifications and uses the model-checking tool* FDR*. It is realized by combining methods for the development as well as for the analysis. It is illustrated by examples and results obtained during the verification of the Byzantine agreement protocol implementation, where the combination of different abstraction methods is required.*

1 Introduction

The acceptance of Formal Methods in industry especially 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. Some experiences in this aspect were gained during our investigation of the deadlock and livelock freedom of a fault-tolerant system designed and implemented by DaimlerChrysler Aerospace for the International Space Station ISS, see [2, 3]. In this paper we will report on an new aspect – the combination of abstraction methods for protocol specification and verification. These methods allow to represent systems by hiding details which are unnecessary for the specific verification objective, thereby reducing the size and complexity of system specifications.

The project started in 1995 when JP Software-Consulting (now Verified Systems International GmbH) in collaboration with the Bremen Institute for Safe System (BISS) were contracted by DaimlerChrysler Aerospace (DASA) to perform an analysis of a fault-tolerant data management system (DMS-R) for the ISS. In addition to data management services provided for experiments carried out in the ISS, DMS-R is also used for flight control during space station assembly and re-boost operations.

During this project, the following correctness aspects of DMS-R have been investigated:

- Deadlock freedom was verified using CSP representations of system components [2] (see [6, 16] for the standard references on CSP).
- Livelock freedom was also verified with CSP-based methods [3].
- Code verification of crucial sequential algorithms was performed by using Hoare Logic and reasoning about pre- and post conditions.
- Load analysis to show the absence of bottlenecks was performed using Stochastic Petri Nets [18].
- Automated hardware-in-the-loop tests based on Timed CSP specifications were performed on system level to analyze the proper cooperation between hardware and software [14, 13].

The present paper focuses on the abstraction methods applied in the specification and verification of the Byzantine agreement protocol implementation used to support the fault-tolerance mechanisms in the DMS-R system. These methods differ from the techniques used in the verification of deadlock freedom, livelock freedom and absence of bottlenecks, where the main focus lies on the *number* of communications which may occur during system operation: In the context of protocols, verification focuses on the *sequencing* of communication events and on the *control data* used for protocol management. As a consequence, the verification techniques described here are complementary to the ones presented in [2, 3, 18].

In the following section a brief description of the DMS-R fault management layer is given. Section 3 provides an overview of the methods used for the analysis, and Section 4 describes the verification process where these methods have been applied. The conclusion outlines future work, especially with respect to tool support for similar projects. Due to the confidential nature of the material unfortunately it is not possible in this framework to make available details of the original code or our specification (these are available only in the internal report [17]).

2 Technical Background

2.1 The Fault-Tolerant Computer

The DMS-R overall architecture consists of up to four lanes operating according to the concept of active redundancy (see Figure 1): The lanes perform their tasks in a synchronized way, and state information is kept consistent between lanes. Each DMS-R application runs on all four lanes in parallel; it communicates with other systems in the ISS via MIL-STD 1553 busses.

Below the application layer, each of the four lanes is structured into an application service layer (ASS), a fault management layer (FML), and the avionics interface (AVI). Applications and the ASS reside on a board using a customized SPARC-type CPU specifically manufactured for space applications by Matra. Both FML and AVI reside on separate transputer boards. The AVI is in charge of the MIL Bus protocol handling according to predefined timing slot allocations.

The purpose of the FML is twofold. It provides the interface between the ASS and AVI of one lane, transferring messages from AVI to ASS and vice versa. For communication between ASS and FML, a VME Bus interface is used; communication between FML and AVI is performed on transputer links. It performs the data transfer between lanes, thus allowing communication between the fault management layers of all lanes. This communication is the basis for error detection, error correction, lane isolation (in the case of an unrecoverable error), and lane reintegration. For this inter-lane communication, transputer links are used: The FML of each lane is connected to the FMLs of all other lanes.

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 all ASS and AVI instances of non-faulty lanes get correct(ed) messages from FML.

The software architecture of 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.

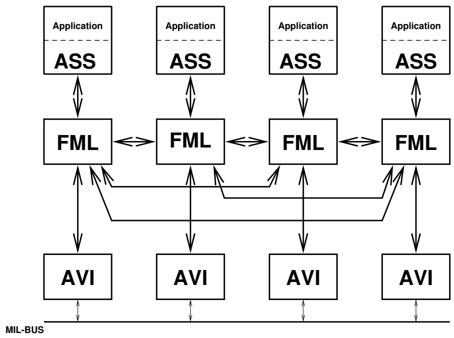

Figure 1. FTC Architecture

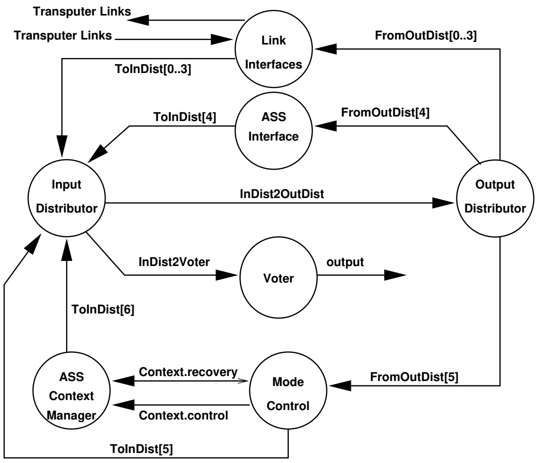

Figure 2. Processes Allocated in Each Lane.

2.2 The Byzantine Agreement Protocol

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behaviour that is often overlooked – namely, sending conflicting information or losing some information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem.

Byzantine Generals Problem. A commanding general must send an order to other three generals called lieutenants such that (1) all loyal lieutenants obey the same order, and (2) if the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

The CSP specification of the Byzantine agreement protocol with four generals and at most one traitor is based on the following *Oral Message Algorithm OM* which can be found in [9].

1. The commander sends his value to every lieutenant.

2. For each i, let v_i be the value Lieutenant i received from the commander, or else be *msgFault* if he receives no value. Lieutenant i sends the value v_i to the other two lieutenants.

3. For each i, and each $j \neq i$ let v_j be the value Lieutenant i received from Lieutenant j in step 2, or else be *msgFault* if he receives no such value. Lieutenant uses the value $majority(v_1, v_2, v_3)$.

The algorithm assumes a function *majority* with the property that the order of the parameters is insignificant for the result.

3 Abstraction Methods

3.1 The Specification Language CSP

The specification language CSP (*Communicating Sequential Processes*) has been designed to describe systems of cooperating processes. CSP processes proceed by engaging into events from a universal alphabet Σ. Processes may be composed by operators which require synchronization on some events. This, rather than assignments to shared state variables, is the fundamental means of interaction between agents.

The CSP operators used in this paper is summarized as follows (see Hoare [6] and the recent book by Roscoe [16] for more details about the CSP syntax).

Operator $\underset{A}{\|}$ is the parallel composition of two processes with synchronization over events in set A; $|||$ describes unsynchronized parallel composition. Operator $\setminus A$ defines abstraction from events contained in set A. Operator $\rightarrow$ is used to specify sequences of event occurrences. Operator $\Box$ denotes external choice which can be controlled by the environment while $\sqcap$ denotes internal choice not to be controlled from the outside. Mutually recursive process equations may be defined by using free process variables X in CSP terms and associate these variables with other process terms. Process STOP blocks everything and never engages into any event.

The specification language CSP is associated with a formal method allowing to verify properties of parallel systems. Formal reasoning is based on the semantics of a process which can be expressed in three mathematical models:

- In the *trace model* a process is represented by the set of finite sequences of events it can perform.

- In the *failures model*, the trace model is extended by associating sets of refusals *Ref* with each trace *tr* of a process *P*: Each *Ref* is a set of events which may be refused by *P* after having performed trace *tr*. The pair (tr, Ref) is called a failure of *P*. Processes which are equivalent in the trace model may be distinguished in the failures model.

- The *failures-divergence model* extends the failures model with the divergences of a process – the traces during or after which the process can perform an infinite sequence of consecutive internal events. Processes which are equivalent in the failures model may be distinguished in the failures-divergence model.

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 can be verified by exhaustive analysis of the transition graph. Moreover, such an analysis can be mechanized. The FDR tool [5] provides this mechanization and has been used for all model checking results described in this document.

Two CSP processes P_1 and P_2 can be compared in model M (trace, failures, failures-divergence model) as follows: We say that P_1 *is refined by* P_2 *in model* M (written as $P_1 \sqsubseteq_M P_2$) if the set of behaviours of P_2 is contained in that of P_1, as far as behaviours can be distinguished in M. For the trace model, $P_1 \sqsubseteq_T P_2$ means that the traces of P_2 represent a subset of P_1-traces. In the failures model, $P_1 \sqsubseteq_F P_2$ implies that each failure of P_2 is a failure of P_1. In the failures-divergence model, $P_1 \sqsubseteq_{FD} P_2$ indicates that both the failures and divergences of P_2 are sub-sets of the respective P_1-sets. All three refinement relations can be verified by model checking, if finite-state specifications are considered. The most nondeterministic non-diverging process *CHAOS* defined as

$$CHAOS(A) = \text{STOP} \bigsqcap (\bigsqcap x : A \bullet (x \rightarrow CHAOS(A)))$$

is refined in the failures model by any processes.

Although failures-divergence refinement implies failures refinement, and failures refinement implies trace refinement, for certain situations trace refinement is easier to deal with than failures or failures-divergence refinement: The investigation of trace refinement properties does not require the analysis of refusal sets which is necessary for failures or failures-divergence refinement checking. Since we have already shown the absence of deadlocks and livelocks in the preceding verification steps, it suffices to verify that the occam implementation refines the Byzantine agreement protocol in the trace model.

3.2 Abstract Interpretation

For a specific verification objective it is unnecessary to inspect every detail of the occam code, since only a subset of the programmed statements have impact on the property to be verified. It is therefore possible to generate a CSP specification which represents an abstract version of the original occam process P showing only the amount of detail which is relevant for the verification of a certain occam process property p. In general, a property expression p will refer to the occam syntax, so that p has no direct meaning on CSP-level. However, it is often quite obvious how an analogous property defined for CSP processes should look like. This motivates the following

Definition 1 *Let P be an* occam *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 abstract interpretation for (P, p), if $\mathcal{A}(P)$ satisfies $\mathcal{A}^*(p)$ always implies P satisfies p.*

If a valid abstract interpretation $(\mathcal{A}(P), \mathcal{A}^*(p))$ for (P, p) is available, we may analyze $(\mathcal{A}(P), \mathcal{A}^*(p))$ instead of (P, p). Obviously, we are interested in finding transformations $(\mathcal{A}(\cdot), \mathcal{A}^*(\cdot))$ from occam processes/properties to CSP processes/properties which produce valid abstract interpretations for a large variety of occam process/property pairs. For the verifications of deadlock and livelock analysis (see [2, 3]) and for the proof of trace properties needed in the protocol verification process described here, the following five basic rules can be applied to construct valid abstract interpretations from occam to CSP:

1. An occam process component P_i which does not influence property p can be abstracted by $CHAOS(A)$, where A denotes the set of channels at the interface P_i.

2. Every sequential algorithm whose results do not influence the property p is deleted.

3. Suppose that valid abstract interpretations $(\mathcal{A}(P), \mathcal{A}^*(p))$, $(\mathcal{A}(Q), \mathcal{A}^*(p))$ for two occam processes P, Q and property p are available. Then every occam IF-construct `IF condition THEN P ELSE Q` may be abstracted to **if** *condition* **then** $\mathcal{A}(P)$ **else** $\mathcal{A}(Q)$.

4. Suppose that valid abstract interpretations $(\mathcal{A}(P), \mathcal{A}^*(p))$, $(\mathcal{A}(Q), \mathcal{A}^*(p))$ for two occam processes P, Q and property p are available. Then every occam IF-construct `IF condition THEN P ELSE Q` may be abstracted to $\mathcal{A}(P) \sqcap \mathcal{A}(Q)$.

5. Suppose that valid abstract interpretations $(\mathcal{A}(P), \mathcal{A}^*(p))$, $(\mathcal{A}(Q), \mathcal{A}^*(p))$ for two occam processes P, Q and property p are available. Suppose further, that the interface I between P and Q is lifted in one-one fashion to $\mathcal{A}(P)$ and $\mathcal{A}(Q)$. Then $(\mathcal{A}(P) \underset{I}{\|} \mathcal{A}(Q), \mathcal{A}^*(p))$ is a valid abstract interpretation of P and Q operating in parallel.

6. If occam property p is influenced only by the occurrence of communications on occam channel c but does not depend on the communicated values x, then all occam communications $c!x$, $c?y$ may be abstracted by the single CSP event c.

Using rules 3,4 and 5, larger abstract interpretations can be built from existing ones. Observe that rule 4 is a "coarser" alternative to the abstraction rule 3.

If a valid abstract interpretation $(\mathcal{A}(P), \mathcal{A}^*(p))$ for an occam process/property pair (P, p) does not satisfy $\mathcal{A}^*(p)$, two cases are possible: Either the occam process does not satisfy p – in this case the verification effort has uncovered an error in the implementation of P – or we have introduced too high a degree of nondeterminism in its CSP abstraction $\mathcal{A}(P)$, for example, by abstracting too many occam `IF-THEN-ELSE` constructs to internal choice $\sqcap$. If it can be expected that P really satisfies p, we then have to find another valid abstraction interpretation which is more deterministic than the one we have constructed.

To show the applications of these rules, we give an example including two parallel processes P and Q that communicate through the internal channels a, b, c and have a common variable mc initialized to 0. The property p to be verified is *deadlock freedom of P, Q with respect to channels a, b, c*. The occam definitions of P and Q are:

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

In the first abstraction, the occam IF-construct is simply converted to CSP internal choice $\sqcap$ (see Rule 4 above). Applying Rule 6, channels a, b, c are abstracted to single CSP events a, b, c. We denote the CSP processes abstracting P, Q and their parallel composition by P', Q' and S'_1, respectively. The lifted property $\mathcal{A}^*(p)$ is now *deadlock freedom of S'_1 with respect to* CSP *events a, b, c*.

$$\begin{aligned} P' &= a \to c \to P' \sqcap b \to c \to P' \\ Q' &= a \to c \to Q' \sqcap b \to c \to Q' \\ S'_1 &= (P' \underset{\{a,b,c\}}{\|} Q') \end{aligned}$$

Unfortunately, S'_1 is not free of deadlock: If P' chooses the first alternative, but Q' the second, the system blocks. Since our hypothesis is that the occam process system really satisfies p, we try a more fine-grained abstraction S'_2 (see the following abstraction). Now 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 *read_mc*, *write_mc* implement these operations on the variable *mc*. The occam IF-construct is now interpreted as a CSP if-construct.

$$
\begin{array}{ll}
MC(x) = & read\_mc!x \rightarrow MC(x) \,\Box\, write\_mc?y \rightarrow MC(y) \\
P' = & read\_mc?mc \rightarrow \\
 & \textbf{if } (mc == 1) \textbf{ then } a \rightarrow c \rightarrow P' \\
 & \textbf{else } b \rightarrow c \rightarrow P' \\
Q' = & read\_mc?mc \rightarrow \\
 & \textbf{if } (mc == 1) \\
 & \textbf{then } a \rightarrow write\_mc!0 \rightarrow c \rightarrow Q' \\
 & \textbf{else } b \rightarrow write\_mc!1 \rightarrow c \rightarrow Q' \\
S'_2 = & ((P' \underset{\{a,b,c\}}{\|} Q') \underset{\{read\_mc,write\_mc\}}{\|} MC(0)) \\
 & \setminus \{read\_mc, write\_mc\}
\end{array}
$$

It can be proved – for example by model checking using the FDR tool – that S'_2 is free of deadlock, which implies that the occam process also satisfies these properties.

The above concept of abstract interpretation may be used to "lift" the occam code verification problem into the domain of CSP verification problems. The next abstraction techniques to be introduced are therefore defined in the CSP domain only, with the objective to further simplify the CSP processes to be analyzed in the verification process.

3.3 Abstraction through Refinement

If process P_1 is refined by P_2 in model M, the set of behaviours of P_2 is included in the set of behaviours of P_1 in model M. This may be interpreted as P_1 being more abstract than P_2 in model M, since it hides more details, thereby identifying more behaviours. Moreover, properties of a process P_2 which can be expressed in model M, may always be re-formulated as a refinement property $S \sqsubseteq_M P_2$ of P_2 in relation to a *specification process* S. This motivates the following definition:

Definition 2 *Let S, P_1 and P_2 be CSP processes. P_1 is called an S-valid abstraction of P_2 in M, if $S \sqsubseteq_M P_1$ implies $S \sqsubseteq_M P_2$.*

The following theorem is an immediate consequence of the fact that refinement $\sqsubseteq_M$ is a transitive relation:

Theorem 1 *If $P_1 \sqsubseteq_M P_2$ holds, then for any specification process S, P_1 is an S-valid abstraction of P_2 in M.*

The notion of S-valid abstractions induces the next verification strategy which replaces one complex verification obligation by two potentially simpler ones. If the proof of property $S \sqsubseteq_M P_2$ is infeasible due to the complexity of P_2, proceed as follows: (1) Construct a simpler process P_1 and prove that $P_1 \sqsubseteq_M P_2$. (2) Prove that $S \sqsubseteq_M P_1$ holds.

A crucial property of abstraction through refinement is the fact that it is preserved under composition using the CSP operators. This follows from the fact that refinement is preserved under compositions involving these operators. As a consequence, Theorem 1 may be generalized to

Theorem 2 *Suppose $\mathcal{A}(P_i) \sqsubseteq_M P_i$ holds for $i : 0..n$. Let ω be an n-ary CSP operator. Then $\omega(\mathcal{A}(P_0), \ldots, \mathcal{A}(P_n))$ is an S-valid abstraction of $\omega(P_0, \ldots, P_n)$ for any CSP specification process S.*

3.4 Eager and Lazy Abstraction

There are often situations where only a subset of events $X \subseteq \Sigma$ of a process P is interesting for analysis of certain properties. To capture this situation in a formal way, the following modes of abstraction have been introduced in [16]:

Definition 3 *Let P be a process free of livelock, H a subset events of P. Then the eager abstraction of P with respect to H is defined as $\mathcal{E}_H(P) =_{df} P \setminus H$ and the lazy abstraction as $\mathcal{L}_H(P) =_{df} (P \underset{H}{\|} CHAOS(H)) \setminus H$.*

Although the eager mode is rather limited as a means of abstraction, it can be used in the trace model as a simple way to isolate irrelevant channels from systems.

Theorem 3 *Let $X = (Q \underset{H}{\|} R) \setminus H$ be a process and H the set of events R can perform. Then for any specification process S, (1) the eager abstraction of Q with respect to H is an S-valid abstraction of X in the trace model, and (2) the lazy abstraction of Q with respect to H is an S-valid abstraction of X in the failures model.*

Part (1) of the theorem follows from the fact that the traces of $X = (Q \underset{H}{\|} R) \setminus H$ are a subset of the traces of $Q \setminus H$. Part (2) follows from the fact that refinement is compositional and the most nondeterministic process *CHAOS* is refined by any processes in the failures model.

3.5 Abstraction through Data-Independence

Many of the applications CSP is best suited for are systems that handle data items without altering them. Rather they manage data values, and input and output them – hopefully at appropriate times at the desired channels. Some of the most striking examples are communication protocols. Consider as an example the following well-known processes We want to show that: $COPY \sqsubseteq_{FD} SYSTEM$.

$$
\begin{array}{lll}
COPY & = & left?x \rightarrow right!x \rightarrow X \\
SEND & = & left?x \rightarrow mid!x \rightarrow ack \rightarrow SEND \\
RECV & = & mid?x \rightarrow right!x \rightarrow ack \rightarrow RECV \\
SYSTEM & = & (SEND \underset{\{mid,ack\}}{\|} RECV) \setminus \{mid, ack\}
\end{array}
$$

COPY and *SYSTEM* might have infinite data values and hence infinitely many states, but they only input and output the values along the channels — the particular value read from a channel influences its future behaviour only as far as its output is concerned, i.e. their behaviours are independent of these values. Hence, we should be able to model-check the refinement $COPY \sqsubseteq_{FD} SYSTEM$ with a single "dummy data value" in place of the infinite set of values, and the result should be generally valid.

This observation is closely related to the recently developed theory of *data-independence analysis* [16, 11]. Lazić ([10], cited in [16]) gives several criteria for processes under which properties for these processes are independent of the size of a type T (which can be the type of a parameter of P, or one of its channels, etc.) provided T is larger than a certain threshold. In other words, one can prove a result for all sufficiently large T via one check on FDR. The theory is still very complex, we will just present some related conclusions.

A data type T is treated by a process P *data-independently*, if P inputs and outputs values of type T along its channels, stores them for later use, but never performs any computations (like addition or evaluation of set membership) constraining what T might be. We say that a data-independent process P satisfies condition *NoEqT* when it contains no test of equality between members of T. The following theorem is derived from [16, Theorem 15.2.1].

Theorem 4 *Suppose we have processes SPEC and IMPL, each data-independent with respect to the type parameter T and additionally satisfying condition NoEqT. Then a one-element type is a valid abstraction of T for verifying the refinement relation* $SPEC \sqsubseteq_T IMPL$.

Equality tests over a data type T often occur in systems with redundant components for decisions. A process P satisfies condition *Norm* in the trace model if it contains neither hiding or renaming nor sequential or interrupt operator. Additionally, let

- W^P denote the maximum number of values of type T that the process S ever has to store for future use,
- $L^P_?$ be the largest number of values of type T that can be input in any visible event of P, and
- $L^P_{\sqcap}$ the largest number of values of type T that can be nondeterministically chosen in any single nondeterministic choice mode over sets involving T in the process P.

Following Theorem 15.2.3 in [16] we have

Theorem 5 *Suppose processes* SPEC *and* IMPL *are data-independent with respect to the type parameter T,* SPEC *satisfies the condition Norm. Let*

$$m = W^{\mathsf{SPEC}} + W^{\mathsf{IMPL}} + \max(L^{\mathsf{IMPL}}_?, L^{\mathsf{IMPL}}_{\sqcap}),$$

then an m-element type is a valid abstraction of T for verifying the refinement relation $\mathsf{SPEC} \sqsubseteq_T \mathsf{IMPL}$.

4 The Verification Process

The overall goal of our verification is to show that the occam code of the FML implements the Byzantine agreement protocol correctly. Since we already know from previous verifications that the FML is free of deadlocks and livelocks, it suffices to prove the property *byzan* stating that every possible communication trace on occam level is consistent with the protocol. We present the principles of three major verification steps which are part of the full protocol verification suite, where each step illustrates one of the abstraction principles introduced above.

Step 1: Abstract Interpretation

The first verification step applies the abstract interpretation technique with the objective to lift the distributed occam program and property *byzan* to a valid abstract interpretation ($\mathsf{FML}, byzan^*$), where FML is a CSP representation of the fault management layer and $byzan^*$ states that every CSP trace FML can perform is a valid trace of the Byzantine protocol specified in CSP. CSP property $byzan^*$ can be expressed as a refinement property $BYZAN \sqsubseteq_T$ FML, where *BYZAN* is a CSP specification process (not shown here due to the usual page limitations) of the algorithm *OM* (see 2.2) for the Byzantine agreement protocol.

The abstract interpretation FML below describes the situation where lane 3 is the commanding lane and lane 0, 1 and 2 are lieutenant lanes. If we can prove that this situation refines *BYZAN* correctly, the same holds for the other situations with lanes 0,1,2 as commander, because the occam protocol implementation is completely symmetric with respect to the lane number. The transputer links are denoted by CSP channels link. Process Lane(i) models the full FML layer in one lieutenant lane i, with CSP sub-processes corresponding to the ones shown in Figure 2. CSP process Commander models the commanding lane; it only sends commands to other lanes and does not vote; its specification is omitted here.

$$\begin{aligned}\mathsf{FML} = \quad & (\mathsf{Commander} \underset{I_1}{\|} \\ & ((\mathsf{Lane}(0) \underset{I_2}{\|} \mathsf{Lane}(1)) \underset{I_3}{\|} \mathsf{Lane}(2))) \\ & \setminus (I_1 \cup I_2 \cup I_3)\end{aligned}$$

where

$$\begin{array}{ll} I_1 = & \{\mathsf{link}.3.i, \mathsf{link}.i.3 \mid i \in \{0,1,2\}\} \\ I_2 = & \{\mathsf{link}.0.1, \mathsf{link}.1.0\} \\ I_3 = & \{\mathsf{link}.1.2, \mathsf{link}.0.2, \mathsf{link}.2.1, \mathsf{link}.2.0\} \end{array}$$

Next we present the decomposition of the processes Lane(i): Sub-processes InDist, OutDist, Link_If, Voter, ASS_If, ModeCo and ASSContext are abstract interpretations of the corresponding occam processes shown in Figure 2. For the detailed CSP specifications of these processes, we have applied the abstraction rules listed in Section 3.2; therefore it is a simple routine verification to prove that the abstraction $(\mathsf{FML}, \mathit{byzan}^*)$ is valid for the occam FML and property *byzan*.

$$\begin{array}{ll} \mathsf{Lane}(i) = & (((((((\mathsf{InDist} \underset{I_4}{\|} \mathsf{OutDist}) \setminus I_4) \\ & \underset{I_5}{\|} \mathsf{Link\_If}(i)) \setminus I_5) \underset{I_6}{\|} \mathsf{Voter}(i)) \setminus I_6) \underset{I_7}{\|} \\ & (\mathsf{ASS\_If} \,|||\, ((\mathsf{ModeCo} \underset{I_8}{\|} \mathsf{ASSContext}) \setminus I_8)) \\ &) \setminus I_7 \end{array}$$

where

$$\begin{array}{ll} I_4 = & \{\mathsf{InDist2OutDist}\} \\ I_5 = & \{\mathsf{ToInDist}.i, \mathsf{FromOutDist}.i \mid i \in \{1,2,3\}\} \\ I_6 = & \{\mathsf{InDist2Voter}\} \\ I_7 = & \{\mathsf{ToInDist}.i, \mathsf{FromOutDist}.j \\ & \mid i \in \{4,5,6\}, j \in \{4,5\}\} \\ I_8 = & \{\mathsf{Context\_recovery}, \mathsf{Context\_control}\} \end{array}$$

Step 2: Abstraction through Refinement

If we could prove directly by model checking with FDR that the abstract interpretation FML introduced above refines the Byzantine protocol specification process *BYZAN*, the verification of the occam implementation would be finished. Unfortunately – as was to be expected – FML is far too complex to be analyzed as one chunk of CSP code by a model checker. Therefore we apply the abstraction by refinement techniques introduced in Section 3.3 on the subprocesses of Lane(i), in order to replace them by less complex components.

For example, a valid abstraction Voter_ABS of the process Voter was constructed, and the following refinement relation was checked with FDR: $\mathsf{Voter\_ABS} \sqsubseteq_T \mathsf{Voter}$. Whereas Voter consists of two communicating parallel subprocesses, Voter_ABS is a sequential process without internal communications.

While all channels between FML main processes are important for deadlock and livelock analysis, only channels in I_7 of a lieutenant are irrelevant considering the Byzantine agreement protocol.

Indeed, the specification of a lieutenant lane may be restructured by two intermediate parallel processes. Process REST describes the behaviours which are irrelevant to the verification of the Byzantine agreement protocol.

$$\begin{array}{ll} \mathsf{Lane}(i) = & (\mathsf{BP\_IMP}(i) \underset{I_7}{\|} \mathsf{REST}) \setminus I_7 \\ \\ \mathsf{BP\_IMP}(i) = & (((((\mathsf{InDist} \underset{I_4}{\|} \mathsf{OutDist}) \setminus I_4) \\ & \underset{I_5}{\|} \mathsf{Link\_If}(i)) \setminus I_5) \underset{I_6}{\|} \mathsf{Voter}(i)) \setminus I_6 \\ \\ \mathsf{REST} = & \mathsf{ASS\_If} \,|||\, ((\mathsf{ModeCo} \underset{I_8}{\|} \mathsf{ASSContext}) \setminus I_8) \end{array}$$

As a consequence, we can make the following lazy abstraction of process BP_IMP and use it at a later stage:

$$\mathsf{Lane\_ABS}(i) = (\mathsf{BP\_IMP}(i) \underset{I_7}{\|} \mathit{CHAOS}(H)) \setminus I_7$$

Step 3: Data-Independency Analysis

Channels $\mathsf{ToInDist}.\{0,\ldots,6\}$ are inputs of process InDist. Channel numbers $\{0,4,5,6\}$ are for lane-local messages from AVI, ASS_If, ModeCo and ASSContext. Messages from the other three lanes are received on channels $\mathsf{ToInDist}.\{1,2,3\}$. We can treat ToInDist as two independent channels: $\mathsf{ToInDist}_1 : \{1,2,3\}$ and $\mathsf{ToInDist}_2 : \{0,4,5,6\}$.

Since both the CSP specification of the Byzantine agreement protocol and the abstract interpretation of the occam implementation are data-independent with respect to the type of $\mathsf{ToInDist}_1$ and contains no test of equality between members of the type, a one-element type is sufficient as an abstraction of the type. We can do the same with the type of $\mathsf{ToInDist}_2$, and decide to take a two-element (one for local messages, and one for messages from other lanes) type as an abstraction for channels ToInDist.

Messages to be voted belong to a type with quasi-infinite domain, but model-checking tools like FDR can only treat process with finite states and finite channels. Abstracting the type of messages is indispensable. Fortunately, both the specification and the implementation are data-independent with respect to the type of messages, and the specification satisfies the condition *Norm* introduced in Section 3.5. As a consequence, Theorem 5 can be applied with

$$W^{\mathsf{SPEC}} = 3,\ W^{\mathsf{IMPL}} = 3,\ L_?^{\mathsf{IMPL}} = 1,\ L_{\sqcap}^{\mathsf{IMPL}} = 0.$$

As the result we could take a seven-element type as an abstraction of the message type.

5 Conclusion

The main obstacle for the verification real-world industrial systems is their size and complexity. Without suitable approaches for splitting the task into manageable subtasks and for reducing the complexity of each intermediate verification objective the attempt to use formal methods and related tools is futile. For the field of protocol verification, we have shown that abstraction methods, such as

- abstract interpretation of occam code in CSP specification,
- abstraction from irrelevant events through refinement,
- abstraction from independent data through data-independent analysis,

offer a powerful means to overcome these size and complexity limitations.

While there exists support for model checking refinement relations – for our purposes, the FDR model checker was a suitable tool – the other abstraction methods had to be applied in a manual way. Future work will comprise the integration of existing tools or the development of new ones to support these tasks.

It should be emphasized that our abstract interpretation method does not depend on occam as an implementation language: Any programming language extended by synchronous communication concepts – such as the ADA rendezvous, C programs using the synchronous communication services of the QNX real-time operating system, or Java multithreading programs – can be lifted to CSP in a convenient way.

Finally, it will be useful to classify abstraction methods with respect to their specific properties in order to develop heuristics for their application in the context of various verification objectives. Additionally, formalizing the abstraction techniques is a necessary effort to gain some tool support and to justify the correctness of their application. For this type of justification, we plan to employ tools like Isabelle or HOL. Some initial work in this direction can be found in [1, 8].

References

[1] B. Buth, R. Cardell-Oliver, J. Peleska: Combining tools for the verification of fault-tolerant systems. In R. Berghammer, B. Buth, J. Peleska (eds.), *Tools for Software Development and Verification*, volume 1 of *BISS Monographs*. Shaker-Verlag, 1998.

[2] B. Buth, M. Kouvaras, J. Peleska and H. Shi: Deadlock analysis for a fault-tolerant system. In M. Johnson (ed.), *Algebraic Methodology and Software Technology. Proceedings of the AMAST'97*, LNCS 1394, December 1997.

[3] B. Buth, J. Peleska and H. Shi: Livelock analysis for a fault-tolerant system. In A. M. Haeberer (Ed.): *Algebraic Methodology and Software Technology. Proceedings of the AMAST'98*, LNCS 1548, December 1998.

[4] N. Dathi: *Deadlock and Deadlock-Freedom,* Oxford University, D. Phil Thesis,1990.

[5] Formal Systems: *Failures Divergence Refinement* FDR*2* Preliminary Manual. Formal Systems (Europe) Lts (1995).

[6] C.A.R. Hoare: *Communicating Sequential Processes.* Prentice-Hall International (1985).

[7] G.J. Holzmann: *Design and Validation of Computer Protocols.* Prentice-Hall Software Series (1991).

[8] Kolyang, C. Lüth, T. Meyer and B. Wolff: Generic Interfaces for Formal Development Support Tools. In R. Berghammer, B. Buth, J. Peleska (eds.), *Tools for Software Development and Verification*, volume 1 of *BISS Monographs*. Shaker-Verlag, 1998.

[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] R.S. Lazić: *Theories for mechanical verification of data-independent CSP*, Oxford University Computing Laboratory technical report, 1997.

[11] R.S. Lazić, and A.W. Roscoe: *Using logical relations for automated verification of data-independent CSP,* Proceedings of the Workshop on Automated Formal Methods (Oxford), Electronic Notes in Theoretical Computer Science 5, 1997.

[12] M. R. Lyu (ed.): *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, Computing McGraw-Hill

[13] J. Peleska and M. Siegel: Test Automation of Safety-Critical Reactive Systems. *South African Computer Journal* (1997)19:53-77.

[14] J. Peleska and C. Zahlten: Test Automation for Avionic Systems and Space Technology (Extended Abstract). Softwaretechnik-Trends (1999)19:34-36.

[15] A. W. Roscoe: *CSP and determinism in security modelling*. In: IEEE Symposium of Security and Privacy, (1995).

[16] A. W. Roscoe: *The Theory and Practice of Concurrency* Prentice-Hall International (1998).

[17] H. Shi and J. Peleska: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Fault Management Layer (FML): Verification of Byzantine Agreement Protocol Implementation.* Technical Report, JP Software-Consulting, (1998).

[18] L. Twele, H. Schlingloff and H. Szczerbicka: Performability Analysis of an Avionics-Interface; Proc. IEEE Conf. on Systems, Man and Cybernetics; San Diego, N.J., pp. 499-504, (Oct. 1998)