Formal Methods in Robotics: Fault Tree Based Verification

Axel Lankenau and Oliver Meyer

Bremen Institute of Safe Systems
TZI, University of Bremen
P.O. Box 330440, D–28334 Bremen
{alone, emm}@tzi.de
phone: +49-421-218-{4684, 3337}
FAX: +49-421-218-3054

Abstract. The intention of this paper is to emphasize the importance of employing formal methods for the design of robotic systems. After a brief survey of current research in this area, a set of requirements is discussed that formal development process should fulfil. As an extended example, a general verification approach for reactive systems is described in detail. It is based on a CSP specification of a fault tree that observes the behaviour of the target system. A template for the modelling of fault tree leaves and nodes is given, and it is instantiated by a "real–world" application taken from the field of mobile robotics.

Keywords: Safety in Robotics, Formal Verification, Fault Tree Analysis
Target Audience: Technical
Basis: Academic Research

1 Introduction

When considering systems engineering tasks in the context of avionics or railway interlocking systems, the use of formal methods in the design process is becoming state of the art. Due to the fact that a malfunction of such systems may cause severe harm to human beings or result in other catastrophic consequences, these systems are referred to as *safety–critical systems* [27, 16]. Formal methods are used to formally specify the safety requirements of these systems, to define the physical model of the system and its environment. They facilitate the verification, validation and test (VVT) processes and are absolutely necessary if these processes are to be automated.

As the potential damage robot actions can cause to human beings increases with the spatial proximity in which man and machine operate, service and especially rehabilitation robots have to be classified as safety–critical systems.

This point of view represents an extension to a common understanding in the robotics community that a *reliable* robot is a *dependable* robot (for definitions cf. [14]). This statement may be true for exploration robots which operate in volcanoes or on Mars, but it is an inadequate simplification of matters with respect to those robots that work in the vicinity of men. For them, the notion *dependable* additionally comprises the properties *available* and *safe*.

There exists a lot of knowledge how safety–critical systems such as nuclear power plants or airplanes have to be built. Various requirements engineering methods, a huge variety of specification languages, many verification approaches as well as test tools have been developed. This paper intends to specify the demands that applications in robotics impose on a formal method to be employed in this domain. For the first time, a method which comprises hazard analysis, the formal specification of safety requirements and their subsequent verification is applied to a service robotics system.

2 Formal Methods in Robotics — A Brief Survey

In contrast to other application fields such as avionics, the use of formal methods is not yet very widespread in the new research areas of service and rehabilitation robotics. This will change in the future for two reasons: on the one hand, most robotic systems are "interestingly complex", i.e. they are more demanding than academic toy examples but are not too extensive to handle. On the other hand, it is commonly agreed that robots such as human power extenders [10] or intelligent wheelchairs [2, 19, 20] should be treated as safety–critical systems.

However, up to now there are only a few research groups which concentrate on the topic of using formal methods to ensure safety properties for (service) robots.

At Lancaster University various approaches to safety analysis were examined. In [24, 26] the concept of a "safety manager" was introduced. An autonomous excavator serves as an application example in these papers. They focus on system analysis topics (development of the hazard analysis method CLASH), and do not cover the verification and validation process [23, 25, 11].

Zhang employed a far more formal approach by developing a semantic model for dynamical systems, the so-called "constraint nets" (cf. [30]). In addition, she proposes a formal verification concept.

The research group "Robots for Hazardous Environments" at Rice University deals with fault–tolerant robot architectures (e.g., cf. [15]). This work aims at improving the reliability of robots through software engineering methods such as fault–tree analysis or risk analysis [6]. Formal verification is not covered.

3 Requirements for Formal Methods in Robotics

Due to the complexity of robotic applications mentioned, the employed formal methods should cover several different problem domains.

Robotics applications are always *hybrid real-time systems*, e.g. industrial robots have to control the angles of joints, mobile robots deal with speeds and braking distances etc. Therefore, the employed formal methods for specification, validation, verification and test have to cope with the combination of discrete events and continuous values and their effects on each other. Furthermore, it must be possible to specify and analyse hard real-time properties. Most tool-supported formal methods used today are only capable of handling discrete systems.

Robots, like all reactive systems, are only able to operate correctly if the environments they are operating in fulfil certain conditions. The specifications of these conditions are the basis for the VVT phases. It is desirable that the formal methods not only allow to take these requirements on the environment into account but provide methods to deduce them during the specification phase of the system.

The complexity of robotics systems requires hierarchical methods which allow the stepwise refinement from abstract to more concrete properties. Otherwise it would be almost impossible to check the completeness of specifications or correctness proofs. Furthermore, hierarchical methods are usually compositional. This makes it possible to easily re-use parts of specifications for similiar problem domains. Since the high levels are only describing abstract properties of robots in general without details about the specific hardware or the intended implementation, their re-use reduces the necessary effort to argue about the completeness of specifications for new projects. Besides, compositionality of the specification method might enable the partial re-use of proofs or proof scripts for these developments.

In order to simplify the handling of the previous aspects it is essential to support the applied formal methods by tools which cover the whole development cycle. It should be possible to use the results of the specification phase for the whole VVT process without having to rewrite them in different languages or formalisms.

The formal approach applied in this paper focusses on the aspects of (interval based) hard real-time properties, deduction of environment specifications and compositionality/re-usability. The integration of *general* real-time requirements as well as complete support for hybrid systems is part of our current work.

4 Fault–Tree Based Verification in CSP — An Extended Example

In the sequel, a new verification method is introduced and instantiated by a service robotics example. It is based on a fault tree documenting the result of a structured hazard analysis. The fault tree specifies threats the system is exposed to and makes it possible to consistently define safety requirements that the operational system must never violate. By modelling the fault tree in CSP it becomes the reference specification of undesired behaviour the system is verified against. Verification is accomplished by model checking to ensure that the top hazard specified within the fault tree hierarchy never occurs.

On the other hand, the same fault tree may serve as a basis for test evaluation if an already implemented system is to be automatically tested. In this case, the

I/O behaviour of the system and the environment can be modelled as a sequence of events, and thus becomes able to trigger the implemented fault tree leaves, causing the propagation of faults through the hierarchy of hazards.

As the example presented below is part of a robotics application (the Bremen Autonomous Wheelchair, see [13]), another subtle advantage of the fault tree based verification is worth mentioning. In robotics, most research projects confine themselves to do mere existence proofs, i.e. it is only shown that a certain robot performs a specific task. In most cases, there is no detailed information given with respect to the circumstances and the environmental conditions under which the experiments were carried out. As indicated above, an additional outcome of the verification method described here is a detailed set of hypotheses specifying the requirements the environment has to satisfy in order to allow the robot to behave as intended.

Besides providing the benefits "one reference document for verification and automated testing (cf. section 5.2)" and "specification of the environment", the approach presented here may be easily adapted to various other problem domains in which embedded controllers are developed.

4.1 Fault Tree Based Verification

Fault tree analysis allows a hierarchically structured and complete derivation of the hazards of a system that has to satisfy safety properties (cf. [28], [7]). It leads to a fault tree that gradually links abstract to more specific threats. The leaves of the tree describe the basic hazards that have to be taken into consideration in system design.

In this paper a graphical representation of fault trees is used that is based on the "alternative symbols" from [9]. Leaves or nodes which are on the same level are a refinement of the higher hazard. The higher hazard occurs if and only if all lower hazards occur (logical AND) or if one of the lower hazards occurs (logical OR), respectively. Note that the occurrence of a hazard does not necessarily mean that the corresponding catastrophy happens, it only means that it is possible to happen. In order to be able to refer to leaves (depicted as circles) and nodes (depicted either as rectangles or as lozenges), they are hierarchically numbered. Besides, we extended the fault tree symbols by a combination of "triangle" and "lozenge" called "external hazard". It represents a hazard which has to be averted by external safety mechanisms and thus does not have to be considered in detail during the system design. The environment requirements are deduced from the set of external hazards.

Consider the small example shown in Fig. 1. In this fault tree hazard `A` occurs if and only if the lower hazards `B` or `C` occur. Accordingly, hazard `B` occurs whenever basic hazard `D` or `E` occurs. Furthermore, hazard `C` occurs if and only if the basic hazard `F` and `G` exist simultaneously, but hazard `G` is an external hazard which is averted by the environment. This leads to an implementation where the developed system may only operate correctly in environments providing the necessary safety mechanisms to avoid the corresponding hazard `G`.

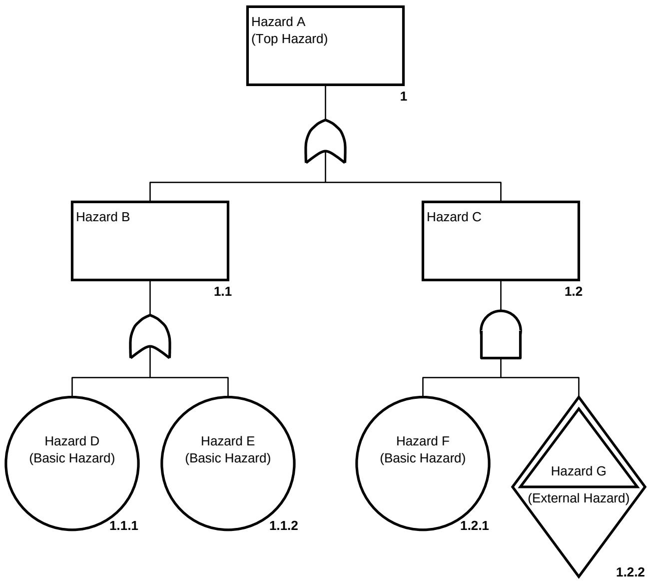

Fig. 1. *A simple fault tree example*

Deriving Safety Requirements from Fault Trees. When using fault trees as a decomposition of the threat space of a system, the usual procedure is to derive *safety requirements*. These are simply negations of the basic hazards. Ensuring that these safety requirements hold is equivalent to ensuring the non-occurrence of the corresponding hazard. Note that it is sufficient for AND–connected hazards if only one of the safety requirements holds to prevent the occurrence of the higher hazard. Besides, it may sometimes be possible to avert higher level hazards and thus one does not has to bother with the more specific hazards in these cases.

The next step is to develop *safety mechanisms* which ensure that the corresponding safety requirements always hold. This has to be realized by taking into account the fault tree hierarchy. It is the task of the safety mechanisms to prevent the environment and a naive realization of the system from performing undesired sequences of events, i.e. it has to block those sequences that may lead to a catastrophic system state. In the sequel, *equipment under control (EUC)* is referred to as representing a simple implementation of the desired functionality without taking into consideration the safety requirements, whereas *system* means the combination of EUC and safety mechanisms.

Since the development of the safety mechanisms cannot be done automatically, it may be erroneous or incomplete. Therefore, a verification is necessary to guarantee that the safety mechanisms are correct with respect to the acceptable faults[1], i.e. to guarantee that the top hazard in the fault tree can never occur (see Fig. 2). Our approach to perform this verification is outlined in the following section.

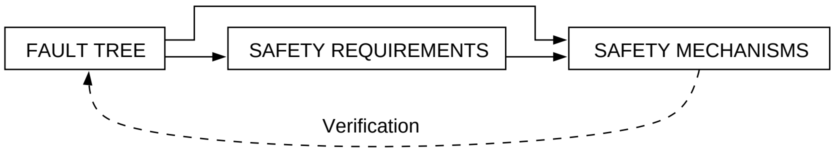

Fig. 2. *Dependencies between fault tree, safety requirements and safety mechanisms. The solid arrows represent a "is derived from"-relation. The process for the derivation of safety mechanisms may be erroneus and therefore a verification is needed.*

It is worth mentioning that the approach chosen does not ensure necessarily safety with respect to *all* possible hazards, but only with respect to those specified in the initial reference document. Here, this document is the fault tree. As its development supports complete coverage of the threat space, it is an adequate means to get the first specification as complete as possible.

Verification Approach. Since the fault tree describes the undesired system behaviour it can be used to verify the correctness of the derived safety mechanisms. It has to be guaranteed that the system can detect and/or treat the basic hazards appropriately to avoid the occurrence of the top hazard in the fault tree.

The basic idea is to use an implementation of the fault tree that runs in parallel with the system and the environment and observes their behaviour. Here, *implementation* of the fault tree refers to a formal specification of the fault tree that can be symbolically executed during the verification process. Whenever a fault leads to a basic hazard it is propagated up the tree according to the specified hierarchy and logic connections between related leaves and nodes. If the propagation continues up to the root the system failed.

In order to ensure that the system *always* behaves correctly it is necessary to examine all possible interleavings of the system components and the environment. This can be done by using a model checker on a formal specification of the safety mechanisms, the environment and the fault tree. Alternatively, classical verification methods could be employed, but they lack the ability to completely automate the verification process.

Both the environment specification and the safety mechanism specifications can trigger exceptions which correspond to the occurrence of a basic hazard.

[1] If hazards are AND–connected the occurrence of single faults can be accepted because it does not lead to the higher level hazard.

These signals are called "artificial events" since they can often be provided by the model of the environment only but not by the real environment. For example, consider a sonar sensor: A model could nondeterministically decide to measure a wrong distance and signal it to the fault tree but a real sensor has no means of detecting it. Besides, it is possible that the safety mechanisms themselves are incorrect. They might introduce new threats and therefore they, too, must be able to trigger artificial events (see Fig. 3).

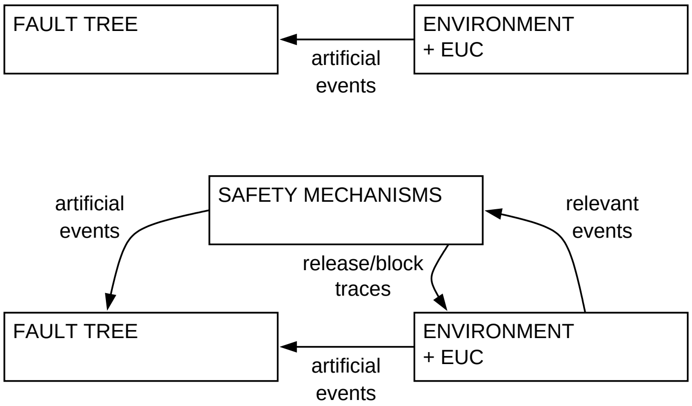

Fig. 3. *The basic idea of the fault tree based verification approach. The environment, the equipment under control (EUC) and the safety mechanisms can trigger artificial events which are observed by the fault tree. In the figure, the upper composition without safety mechanisms can be used to analyse the system behaviour and to demonstrate that the safety mechanisms are necessary. In contrast, the lower composition is employed to verify the non-occurrence of the top hazard.*

4.2 Fault Tree Implementation in CSP

When taking into account that the verification of the safety properties mentioned above is carried out by model checking, it becomes obvious that a formal specification of the fault tree is needed. It is especially important to model the propagation of fault–triggers from the leaves to the root of the tree. As this inherently includes the modelling of communicating processes, the formal specification language CSP (cf. [8] and [21] for syntactic extensions) is used.

An additional reason to choose CSP was the model checking tool FDR (cf. [3]) which allows verifying various refinement relations between specified processes. The template specifications of a leaf, an OR–node and an AND–node are represented in FDR syntax in Figs. 4, 5 and 6, respectively. Thus, it is possible to directly re-use the depicted code segments in other applications.

The basic idea used here is a kind of discrete time slicing. It is assumed that the system works in a loop lasting, say, t time units. During this period of time, the environment and the system may cause faults, i.e. trigger those artificial events the leaves of the fault tree engage in. Meanwhile, the implementation of the fault tree merely observes, i.e. its leaves never refuse the artificial events triggered by the system or the environment. This behaviour is absolutely essential, because the fault tree is a "virtual observer" that must not have any influence on the system or the environment. The t time units period of system activity and fault tree observation is followed by a period of fault tree activity. This period lasts 0 time units, i.e. it happens in nonexisting time. Therefore, the modelled system does not do anything during this period, since it does not exist in reality but is only considered during the verification by model checking.

By using this time slicing technique, it becomes possible to model AND–nodes of the fault tree, because in this way the interval of time during which the hazards exist can be expressed. Furthermore, this approach makes it possible to reinitialize the state of the fault tree in a deterministic way. Usually, systems are prone to cause certain faults only for a certain period of time. These faults either occur within this certain period or they *never* occur. As the system considered here runs in a loop, it is adequate to define the time slice for the system as one turnaround of the loop. In general, it has to be determined for each leaf how long this particular threat lasts, but in the example presented in this paper the global time slicing approach turned out to be sufficient.

In the CSP implementations the time slicing is modelled by the timing events *begin* and *end* as follows: The time between a *begin* and an *end* is the time of system activity and fault tree observation. The time between an *end* and the following *begin* is used by the fault tree implementation to propagate the faults up the tree. Remember, this time period is nonexistent in reality.

In Fig. 4 a template of a leaf implementation in CSP is depicted. The process $LEAF\_N$ represents the nth leaf in the fault tree and waits for the timing event *begin* to occur. This timing event is triggered by the system when it starts a new loop. Together with the *end* timing event and the artificial events triggered by the environment, it builds the synchronized interface between the fault tree and the environment. After engaging in the *begin* event, the leaf accepts either an *end* event (no fault during this loop), or the artificial event art_event_N that triggers leaf n (a fault happened). In case of a fault, the process $CONSUME\_art\_event\_Ns$ "consumes" further occurrences of the artificial event until an *end* signals the end of system activity and beginning of fault tree activity. This is very important because the environment must not be influenced, especially it must not be blocked. Then, the leaf triggers the $signal\_N$ event which causes action in a node in an upper level in the fault tree. If no error occured, the leaf is reset to its initial state. An example instantiation of a leaf will be given in Fig. 10 on page 14.

$$
\begin{array}{l}
LEAF\_N = begin \rightarrow (art\_event\_N \rightarrow CONSUME\_art\_event\_Ns \\
\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\quad end \rightarrow LEAF\_N)
\end{array}
$$

$$
\begin{array}{l}
CONSUME\_art\_event\_Ns = end \rightarrow signal\_N \rightarrow LEAF\_N \\
\qquad\qquad\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\qquad\qquad\quad art\_event\_N \rightarrow CONSUME\_art\_event\_Ns
\end{array}
$$

Fig. 4. *Template of a fault tree leaf.*

As mentioned above, the upper levels of the fault tree consist of OR–nodes or AND–nodes. The template implementation of an OR–node is depicted in Fig. 5. After the initial *begin* has introduced the first turnaround of the system loop, the process OR_NODE_N behaves as follows: It waits for the *end* timing event signalling the start of fault tree activity. If the following event is a *begin*, none of the lower nodes or leaves in the fault tree signalled an error and the process waits for the subsequent *end*. If *any* of the events gathered in the set $failset\_NODE\_N$ happens, one of the lower nodes or leaves has signalled an error. Thus OR_NODE_N itself signals the event *signal_N* triggering a node in an upper level of the fault tree. Again, a "consuming" process is needed in order to cope with cases in which more than one lower node or leaf triggers OR_NODE_N.

$$
\begin{array}{l}
OR\_NODE\_N\_INIT = begin \rightarrow OR\_NODE\_N \\
OR\_NODE\_N = end \rightarrow \\
\qquad (begin \rightarrow OR\_NODE\_N \\
\qquad \Box \\
\qquad \Box_{i:failset\_NODE\_N}\; i \rightarrow signal\_N \rightarrow CONS\_FAILS\_NODE\_N) \\
CONS\_FAILS\_NODE\_N = begin \rightarrow OR\_NODE\_N \\
\qquad\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad\qquad \Box_{i:failset\_NODE\_N}\; i \rightarrow CONS\_FAILS\_NODE\_N
\end{array}
$$

Fig. 5. *Template of a fault tree OR–node.*

The modelling of the AND–nodes (cf. Fig. 6) of the fault tree is slightly more difficult than the one of an OR–node, as only the occurrence of *all* stimulating events of the node (represented as set $failset\_NODE\_N$) causes it to trigger the corresponding node on the higher level. Therefore, during the time period in which the system is active, the AND–node process $AND()$ must accept all stimulating events in arbitrary order. If and only if each event of the initial failset $failset\_NODE\_N$ already occured (i.e. the remaining current failset $failset$ is empty), the upper level of the fault tree is triggered by the $signal\_N$ event. After that, any surplus events are "consumed" by the $CONS\_FAILS\_NODE\_N$ process until the timing event $begin$ sets off a new period of system activity and fault tree observation.

If the current failset is not yet empty, $AND()$ accepts any event of the initial failset. Afterwards, it behaves as if that special event had already occured (cf. $AND(failset - \{i\})$ in Fig. 6).

$$
\begin{aligned}
&AND\_NODE\_N\_INIT = begin \ \rightarrow \ AND\_NODE\_N \\
&AND\_NODE\_N = end \ \rightarrow \ AND(failset\_NODE\_N) \\
&AND(failset) = \\
&\quad (card(failset) = 0) \ \& \ (signal\_N \ \rightarrow \ CONS\_FAILS\_NODE\_N) \\
&\quad (card(failset) > 0) \ \& \ (begin \ \rightarrow \ AND\_NODE\_N \\
&\qquad\qquad\qquad\qquad\qquad \square \\
&\qquad\qquad\qquad\qquad\qquad \square_{i:failset\_NODE\_N} (i \ \rightarrow \ AND(failset - \{i\}))) \\
&CONS\_FAILS\_NODE\_N = begin \ \rightarrow \ AND\_NODE\_N \\
&\qquad\qquad\qquad\qquad\qquad \square \\
&\qquad\qquad\qquad\qquad\qquad \square_{i:failset\_NODE\_N} (i \ \rightarrow \ CONS\_FAILS\_NODE\_N)
\end{aligned}
$$

Fig. 6. *Template of a fault tree AND–node.*

Note that this technique of modelling and checking safety properties is more powerful than simply deriving a linear temporal logic (LTL) statement from the *structure* of the fault tree. The latter may only be used to specify static invariants which have to be always fulfilled by the system while our approach can deal with interval based real-time properties as well. When a leaf of the fault tree is triggered it is activated for the *period of one time slice*. If a different AND-connected leaf at the same hierarchy level is activated later but during the same time slice, the higher node will be triggered even if the first hazard does not exist any longer. In this way it is possible to model and check higher-level hazards which are based not only on the logical relations of hazards but also on their time relations. Therefore, the well-known algorithms to verify LTL statements (cf. for example [4]) are unsuitable to check that the highest node of the fault tree can never be triggered, since the fault tree cannot be expressed by an LTL formula.

4.3 Application: A Safety Layer for an Autonomous Wheelchair

The whole safety layer of the Bremen Autonomous Wheelchair has been developed using the approach described in this paper (cf. [12]). The complete fault tree has about 170 nodes and leaves, the highest levels are shown in Fig. 7. An interesting aspect is the decomposition of the hazard "Collision" (`1.1`) which results in the distinction between active and passive collisions. The wheelchair itself is only able to avoid active collisions, i.e. collisions caused by the movements of the wheelchair. Passive collisions, e.g. caused by automatic doors which hit the wheelchair, have to be avoided by external safety mechanisms of the environment (indicated by the external hazard symbol). Therefore, demanding to avert this external hazard is one element of the set of requirements on the environment in which the wheelchair will operate safely. In this case, automatic doors are assumed to be "intelligent" enough to open and close only if this cannot cause a collision. The necessary safety mechanisms can also be designed with the approach described in this paper. The additional fault tree for automatic doors can be considered as part of the refinement of the "passive collision" external hazard.

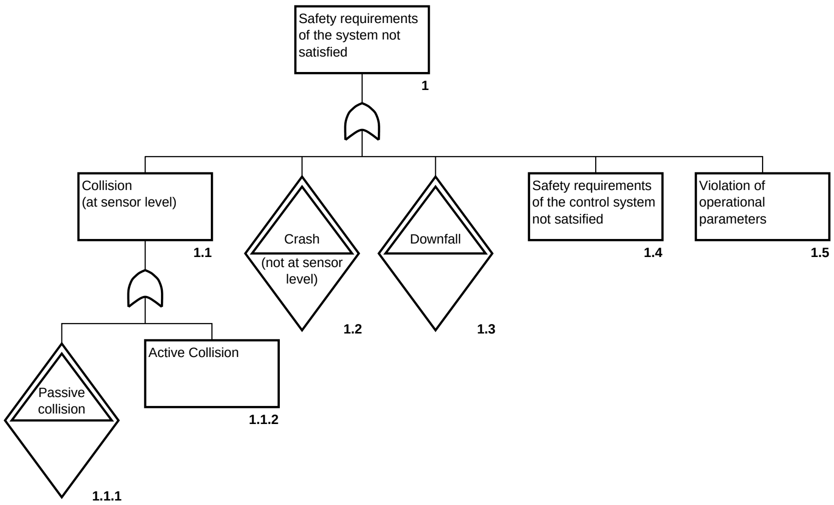

Fig. 7. *The highest levels of the fault tree of the Bremen Autonomous Wheelchair.*

The high levels of the fault tree deal with common hazards that are easy to adapt to other mobile robotics applications. As a consequence, the majority of the verification results might be re-usable as well.

As depicted in Fig. 8 the safety layer developed consist of a sensor/actuator module, net modules and the sensors and actuators of the wheelchair. Since many hardware devices used in robotics can only be used in combination with standard operating systems, it has to take into account that not all components can meet real time restrictions.

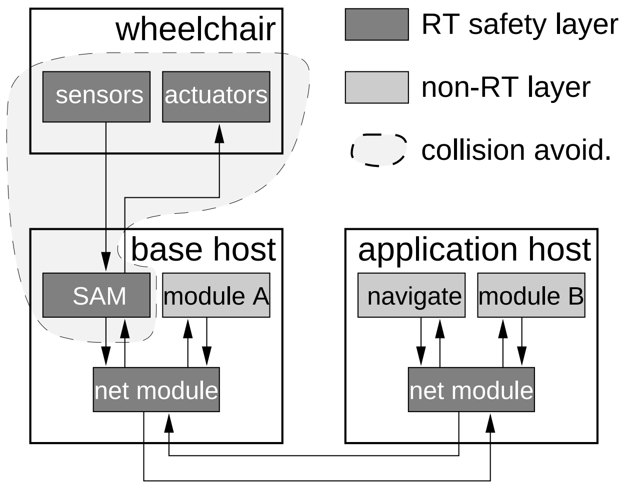

Fig. 8. *The system architecture: the sensor/actuator module (SAM) is the interface to the real wheelchair. The net modules realize a real time communication between the hosts.*

In the context of rehabilitation robotics, not only collisions with obstacles have to be taken into account as a catastrophic system state but also the *non-availability* of the system. For example, for a handicapped person it is essential that an autonomous wheelchair performs the expected task. In case of an emergency, a standstill cannot be considered as a safe system state since the user might have to get to an exit or a telephone. Therefore, reliable communication between basic system components becomes a safety–critical part. In contrast to the collision hazard, the specification of the communication protocol is confined to discrete-value/real-time properties.

Since the safety mechanisms to avert the “active collision” hazard are based on hybrid components and we are not yet able to handle the verification of such hybrid systems, the communication protocol will be analysed in detail in the sequel.

Requirements of the Protocol. In order to guarantee upper time bounds when using a mutli–computer system it is necessary to use a hard real time communication protocol. On the other hand, it is preferable to use cheap standard technology instead of specialized hardware. Therefore, the protocol is realized as an additional layer on top of an ethernet network.

Because ethernet is based on a CSMA/CD protocol the new layer has to avoid collisions on the network. This is realized by a time division multiple access (TDMA) frame–protocol (cf. [29]) which ensures that at any time only one computer can write to the bus. Besides, it is important that every host *period-*

ically has the opportunity to send data. The resulting fault tree segment and its CSP specification is presented in the next section, followed by the model of the environment and a naive realization of the producer and consumer processes (the EUC).

A Fault Tree Segment in CSP. The CSP specification of the fault tree segment for communication problems (see Fig. 9 for an extract) can be realized by using the templates for leaves and nodes shown in section 4.2. The node `X` represents an OR–node which has to agree on fault–triggers from the nodes `X.1` to `X.3`, i.e. the set $failset\_NODE\_X$ contains the events $signal\_X\_1$ to $signal\_X\_3$. Accordingly, the node `X.1` has to be modelled as an AND–node reacting to $signal\_X\_1\_1$ and $signal\_X\_1\_2$.

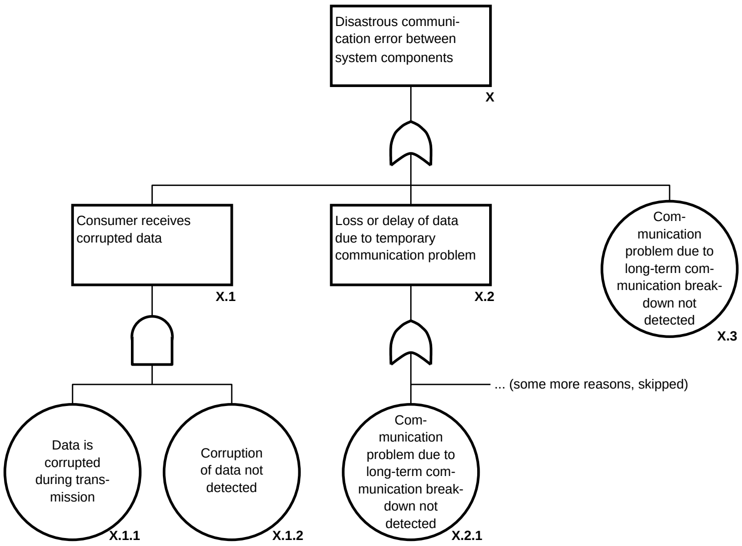

Fig. 9. *Extract of a fault tree segment dealing with communication problems.*

In order to describe the instantiation of a leaf we focus on the basic hazard `X.2.1` — the delay of data might be caused by a collision on the network. Therefore, the model of the network has to provide an artificial event that signals a collision (see section 4.3). The resulting leaf is shown in Fig. 10.

The fault tree process $FAULTTREE$ is composed of node and leaf processes running in parallel. All of them have to agree on the time slicing events *begin* and *end*, the different levels are synchronized by the internal fault tree signals as well. Finally, the whole process has to be synchronized with the system process and the environment by the artificial events to allow the observation of the system

$$
\begin{aligned}
LEAF\_X\_2\_1 = begin \ \rightarrow \ & (art\_event\_Coll \ \rightarrow \ CONSUME\_art\_event\_Cs \\
& \Box \\
& end \ \rightarrow \ LEAF\_X\_2\_1)
\end{aligned}
$$

$$
\begin{aligned}
CONSUME\_art\_event\_Cs = \ & end \ \rightarrow \ signal\_X\_2\_1 \ \rightarrow \ LEAF\_X\_2\_1 \\
& \Box \\
& art\_event\_Coll \ \rightarrow \ CONSUME\_art\_event\_Cs
\end{aligned}
$$

Fig. 10. *Instance of a fault tree leaf in CSP for detecting collisions of data packets on the network.*

behaviour. In order to verify that the system is correct, it is sufficient to show that the highest fault tree node can never trigger the corresponding event (here: $signal\_X$, see Fig. 11). This is done by a refinement check in the trace model against the process $GOODSYS$.

Modelling of the Environment. The specification of the ethernet process shown in Fig. 12 accepts data on the *net_send* channel and passes it to the *net_recv* channel. The *end_transmit* and *end_recv* events model the time it takes to send and receive messages, respectively. Whenever more than one *net_send* event occurs before all participating processes agree on the *end_transmit* event, a collision occurs. This is signalled to the fault tree process by triggering the artificial event art_event_Coll. After a collision has occured all subsequent *net_send* events are ignored.

A simple producer process ($PROD$ in Fig. 13) nondeterministically decides whether to put some data on the network or not. It ignores all activities during the receive phase. Correspondingly, a consumer process always accepts transmissions after the transmit phase has ended.

Drawbacks of a Simple Protocol. It should be pointed out that the environment introduced above is prone to behave maliciously, i.e. it may trigger one of the artificial events that cause the propagation of faults in the fault–tree process $FAULTTREE$ and finally generates the *signal_X* event.

An environment consisting of two producers $PROD(1)$ and $PROD(2)$, a consumer $CONS$ and the underlying network NET behaves faulty if the two producers simultaneously try to transmit data and thus cause a collision of packets on the network. As expected, a trace refinement proves that the sequence of events depicted in Fig. 14 is possible and delivers the highest fault–signal of the fault–tree ($signal\_X$).

The period of system activity starts with the timing event *begin* (cf. 14). Then, each producer generates data and transmits it by engaging in the particular *net_send.x* event. As shown, the second producer transmits its data before

$$
\begin{aligned}
FAULTTREE = {} & ((((LEAF\_X\_1\_1 \\
& \quad \underset{\{begin,end\}}{\|} LEAF\_X\_1\_2) \\
& \quad \underset{\{begin,end\}}{\|} LEAF\_X\_2\_1) \\
& \quad \underset{\{begin,end\}}{\|} ...) \\
& \qquad \underset{\{signal\_X\_1\_1, signal\_X\_1\_2, signal\_X\_2\_1, ..., begin, end\}}{\|} \\
& ((AND\_NODE\_X\_1\_INIT \\
& \quad \underset{\{begin,end\}}{\|} OR\_NODE\_X\_2\_INIT) \\
& \quad \underset{\{begin,end\}}{\|} OR\_NODE\_X\_3\_INIT)) \\
& \qquad \underset{\{signal\_X\_1, signal\_X\_2, signal\_X\_3, begin, end\}}{\|} \\
& OR\_NODE\_X\_INIT
\end{aligned}
$$

$$FTSYS = ESYS \underset{\{art\_event\_Coll, art\_event\_..., begin, end\}}{\|} FAULTTREE$$

$$GOODSYS = CHAOS(\Sigma - \{signal\_X\})$$

Fig. 11. *Construction of the fault tree system $FTSYS$. It can be used to verify that the process $ESYS$ (consisting of the system and its environment) cannot perform traces which lead to the top hazard in the fault tree, i.e. to ensure that the OR–node X never triggers the event $signal\_X$.*

$$
\begin{aligned}
NET = begin \rightarrow {} & (net\_send?data \rightarrow (end\_transmit \\
& \qquad \rightarrow net\_recv!data \\
& \qquad \rightarrow end\_recv \\
& \qquad \rightarrow end \rightarrow NET \\
& \qquad \Box \\
& \qquad net\_send?data \\
& \qquad \rightarrow art\_event\_Coll \\
& \qquad \rightarrow CONSUME\_SENDs; end\_recv \\
& \qquad \rightarrow end \rightarrow NET) \\
& \Box \\
& end\_transmit \rightarrow end\_recv \rightarrow end \rightarrow NET)
\end{aligned}
$$

$$
\begin{aligned}
CONSUME\_SENDs = {} & net\_send?data \rightarrow CONSUME\_SEND \\
& \Box \\
& end\_transmit \rightarrow \mathrm{SKIP}
\end{aligned}
$$

Fig. 12. *The abstract specification of the underlying ethernet network.*

$$PROD(id) = (produce\_Item.id \rightarrow net\_send!id \rightarrow end\_transmit \rightarrow end\_recv \rightarrow PROD(id)) \sqcap (end\_transmit \rightarrow end\_recv \rightarrow PROD(id))$$

$$CONS = end\_transmit \rightarrow (net\_recv?id\_sent \rightarrow end\_recv \rightarrow consume\_Item.id\_sent \rightarrow CONS \;\Box\; end\_recv \rightarrow CONS)$$

Fig. 13. *Specification of producer and consumer processes.*

$begin, produce\_Item.2, produce\_Item.1, net\_send.1, net\_send.2,$
$art\_event\_Coll, end\_transmit, end\_recv, end,$
$signal\_X\_2\_1, signal\_X\_2, signal\_X$

Fig. 14. *Trace leading to the top hazard, indicated by event $signal\_X$*

the data of producer 1 have been received. The resulting collision is signalled by the NET process when issuing the artificial event art_event_Coll. After the period of system activity ended (indicated by the *end* event), the leaf X.2.1 triggers its fault–event. This is propagated up the fault tree until the occurrence of the top hazard $signal\_X$ is signalled.

In order to deal with this problem, the system has to be extended by safety mechanisms, as described in the sequel.

Frame–Protocol with Smart Buffering. As indicated, collisions on the network can be avoided by employing a time division approach, i.e. by strictly assigning the permission to transmit data to only one host at a time. Thus, each attempt to send data will succeed immediately and will not be delayed by the CSMA/CD retransmission technique that is used in case of packet collisions. The frame–protocol is controlled by a dedicated computer in the network, the *master*. Periodically, it requests data of a specific ID by broadcasting. The computer hosting the application that produces such data answers the request by broadcasting the corresponding data to all hosts in the net. This procedure works as long as the data IDs are unique, i.e. it must never happen that two hosts feel responsible to answer the same request.

The applications on the hosts should not be allowed to access the net directly. As, furthermore, it should not be taken for granted that these high level modules meet hard real time restrictions, a double buffering policy is employed. Data received by the net module implemented within the net interrupt is copied into a *net buffer*. During the so–called *housekeeping period* no net activity is allowed and the data from the net buffer is copied to an *application buffer* which is a piece of memory shared between all applications and the net module. Thus, upper time bounds can be defined specifying how long it takes for the data sent by an application to be received by another application on an arbitrary remote host.

To avoid race conditions with regard to access to the shared memory area between the net module and the applications, a procedure is needed to ensure mutual exclusion. As the net module has to access the memory without delay (hard real time conditions), simple locking algorithms fail. This problem is an example for the necessity of iterative fault tree development, since the chosen safety mechanism (buffering with simple locking techniques) causes a new threat (memory access blocked) that has to be taken into consideration in the fault tree. By introducing an artificial event that is triggered by the safety mechanism whenever two applications try to access the same buffer at the same time it can be shown, by model checking, that an extended system including such simple locked buffers fails.

In [12] a smart buffering concept has been developed that allows the net module and all applications to access the shared memory area without delay. The basic idea is to provide *#readers+#writers* times as much memory as usually needed for a buffer, i.e. each reading application accesses its own part of the memory while the net module is able to write into another memory area. It has been verified that this approach ensures mutual exclusion and guarantees the delivery of data within bounded time even if the applications do not meet the real time requirements.

Eventually, it has been shown that the top hazard of the fault tree depicted in Fig. 9 never occurs when employing the frame–protocol in combination with the smart buffering technique.

4.4 Conclusive Remarks

An informal specification of hazards is employed in order to formalize the undesired behaviour of a system and its environment in a hierachically structured fault tree. The fault tree specification is the only reference document required to verify safety properties. It is shown that the chosen approach is an adequate means to prove that a designed safety system can avert all relevant hazards. As segments of the fault tree can be re-used for similar application domains this method facilitates the re-use of verification results.

We are convinced that the integration of informal requirements, the semi–formal fault tree technique and the formal verification using a model checker is one of the strong points of this method. The stepwise increase of the level of formalization makes it possible to describe the requirements in an appropriate way at each stage of development. During the fault tree based hazard analysis it might not be clear what certain informal descriptions exactly mean (*How* close is "close"?). Nevertheless, the formal structure of the fault tree allows a thorough reasoning about the completeness of the specification. When formalizing the hazards of the fault tree to safety requirements and later deriving safety mechanisms, problems may occur resulting from ambigious specifications. The resulting subsequent reiteration of the fault tree analysis often yields important new design criteria, as shown in section 4.3.

5 Conclusion and Future Work

Providing a specification of the environment is a major property demanded from a formal method that is used to develop a safety controller for a robotic application. The presented approach is well-suited to systematically derive environment requirements in combination with safety mechanisms for the developed system. Furthermore, the verification method described here is capable of handling interval-based real-time specifications.

Nevertheless, it does not yet suit arbitrary problem domains, e.g. if hybrid parts are involved. Thus, two extensions are proposed in order to further broaden the applicability of the approach in the future.

5.1 Improved Fault Tree Semantics

The additional incorporation of a fault tree that not only uses a formal structure but also has a formal semantics realized by formally specified leaves, would ease the following deduction of safety requirements significantly (it should be possible to automate this step). However, the informal description of hazards in the first fault tree document should be retained, because it is an excellent base for the communication between the software designer and the customer.

Often, it is desirable to express time dependencies of hazards in the fault tree in a formal way, i.e. an extension of the fault tree syntax and semantics used here that makes it possible to specify time relations between stimulating branches of AND–nodes. Such a semantics would allow to enhance the implementation of the fault tree by introducing a means to assign detailed time periods to specific nodes instead of using a global time slicing in the form of *begin* and *end* events. A similiar approach is presented in [5], where conventional fault trees are formalised and analysed with the minimal cut set approach and with timed petri nets. We will examine if this method can be incorporated into our CSP based verification.

Moreover, facilities for expressing and validating/verifying/testing hybrid aspects of robotics applications are needed. Since the presented verification approach is based on CSP, a compatible hybrid extension should be realized either in Hybrid-CSP (cf. [31]) or with hybrid automata, which can be combined with the CSP world (cf. [1]).

5.2 Robotics and Test Derivation

Many "real–world" applications turn out to be too large to make verification of safety properties possible. Here it is necessary to gain confidence in the system by performing tests. The test cases have to be chosen in a systematic way to ensure that all relevant parts of the system are tested. Therefore, it is desirable to use a test tool, such as RT-Tester (formerly called VVT-RT), that is based on formal methods and selects the test cases so that the test coverage gradually converges to completeness (cf. [17, 18]). One example of such a "real–world" application is the safety layer for the autonomous wheelchair described above. Similar to the verification approach presented here, the fault tree can be used to evaluate its behaviour. Furthermore, the fault tree specifies the acceptable deviations from the normal behaviour of the environment, e.g. the number of consecutive sensor misreadings the system has to cope with. This information can be exploited by the test tool to generate all relevant test cases. Moreover, RT-Tester can already handle the automated test of general hard real-time properties specified in Timed-CSP and it can be used to check the behaviour of hybrid systems (cf. [22]). Thus, the test tool RT-Tester supports all means necessary for the formal *testing* of robotics applications.

References

1. Amthor, P.: A CSP Model for Hybrid Automata. Proceedings of the 3. BCS-FACS Northern Formal Methods Workshop, Ilkley, UK (1998)
2. Bell, D. A., Levine, S. P., Koren, Y., Jaros, L. A., Borenstein, J.: Design Criteria for Obstacle Avoidance in a Shared Control System. Proc. RESNA '94 Conference. Nashville (1994)
3. Formal Systems Ltd.: Failures–Divergence Refinement. FDR2 User Manual, Oxford (1997)
4. Gerth, R. et al.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. Proc. of the Fifteenth International Symposium on Protocol Specification, Testing and Verification, Warsaw (1995)
5. Gorski, J., Wardzinski, A.: Timing Aspects of Fault Tree Analysis of Safety Critical Systems. Proc. of the 5th Safety–critical Systems Symposium. Brighton, England (1997).
6. Hamilton, D. L., Visinsky, M. L., Bennett, J. K., Cavallaro, J. R., Walker, I. D.: Fault-Tolerant Algorithms and Architectures for Robotics. Proc. 1994 IEEE Mediterranean Electrotechnical Conference. Antalya, Turkey (1994).
7. Hansen, K. M.: Linking Safety Analysis to Safety Requirements. PhD Thesis, DTU Technical University of Denmark (1996)
8. Hoare, C. A. R.: Communicating Sequential Processes. International Series in Computer Science, Prentice Hall (1985)
9. International Electrotechnical Commission: International Standard IEC 1025 Fault Tree Analysis (FTA). Geneva (1990)
10. Kazerooni, H.: Human Power Extender: An Example of HumanMachine Interaction via the Transfer of Power and Information Signals. Proc. AMC 98 - 5th Int. Workshop on Advanced Motion Control, Coimbra University Portugal (1998).
11. Kotonya, G., Sommerville, I.: Integrating Safety Analysis and Requirements Engineering. Proc. Joint Asia Pacific Software Engineering Conference (APSEC) and International Computer Science Conference (ICSC) 1997. Hongkong
12. Lankenau, A., Meyer, O.: Der autonome Rollstuhl als sicheres eingebettetes System. Masters Thesis, Universität Bremen (1997). `http://www.informatik.uni-bremen.de/~emm/DA/DA-Abstract.html`
13. Lankenau, A., Meyer, O., Krieg–Brückner, B.: Safety in Robotics: the Bremen Autonomous Wheelchair. Proc. 5th International Workshop on Advanced Motion Control, Coimbra (1998)

14. Laprie, J.C.: Dependability, Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems Vol.5, Springer (1992)
15. Leuschen, M.: Robot Reliability Through Fuzzy Markov Models (MSc Thesis). Rice University, Texas, USA (1997).
16. Leveson, N.: Safeware: System Safety and Computers. Addison-Wesley, Reading, MA (1995)
17. Peleska, J.: Formal Methods and the Development of Dependable Systems. Habilitationsschrift, Christian–Albrechts–Universität zu Kiel (1996). http://www.informatik.uni-bremen.de/~jp/papers/habil.ps.gz
18. Peleska, J.: Testing Reactive Real-Time Systems. Tutorial, held at the FTRTFT 98. Denmark Technical University, Lyngby (1998).
19. Pires, G., Arajo, R., Nunes, U., de Almeida, A.T.: ROBCHAIR – A Powered Wheelchair Using a Behaviour-Based Navigation. Proc. AMC 98 – 5th Int. Workshop on Advanced Motion Control, Coimbra University Portugal (1998).
20. Röfer, T., Lankenau, A.: Architecture and Applications of the Bremen Autonomous Wheelchair. Fourth Joint Conference on Information Systems. Research Triangle Park, North Carolina, USA (1998)
21. Roscoe, A. W.: The Theory and Practice of Concurrency. International Series in Computer Science, Prentice Hall (1998)
22. Schlingloff, H., Meyer, O., Hülsing, Th.: Correctness Analysis of an Embedded Controller. Proceedings of the 10th DASIA Conference. Lisbon, Portugal (1999) (to appear)
23. Seward, D. W., Bradley, A., Margrave, F. W.: Hazard Analysis Techniques for Mobile Construction Robots. Proc. 11th Int. Symp. on Robotics in Construction. Brighton, England (1994).
24. Seward, D. W., Margrave, F. W., Sommerville, I., Kotonya, G.: Safe Systems for Mobile Robots – The Safe–SAM project. Achievement and Assurance of Safety. Proc. Third Safety–critical Systems Symposium. Brighton, England (1995).
25. Seward, D. W., Margrave, F. W., Sommerville, I., Morrey, R.: LUCIE the Robot Excavator – Design for System Safety. Proc. IEEE Int. Conference on Robotics and Automation. Minneapolis, USA (1996).
26. Sommerville, I., Seward, D., Morrey, R., Quayle, S.: Safe Systems Architectures for Autonomous Robots. Proc. Third Safety–critical Systems Symposium, Brighton, England (1995).
27. Storey, N.: Safety–Critical Computer Systems. AddisonWesley (1996).
28. Vesely, W. E. et al.: Fault Tree Handbook. Nuclear Regulatory Commission, Office of Nuclear Regulatory Research, Division of Systems and Reliability Research (1981). ISBN 0-16-005582-2.
29. Tanenbaum, A. S.: Modern Operating Systems. Prentice Hall (1992)
30. Zhang, Y.: A Foundation for the Design and Analysis of Robotic Systems and Behaviors (PhD Thesis). University of British Columbia, Kanada (1994).
31. Chaochen, Z., Ravn, A.P., Hansen, M.R.: An extended Duration Calculus for Hybrid Real-time Systems. In: R.L. Grossman, A. Nerode, A.P. Ravn and H. Rischel (eds.): Hybrid Systems. Springer LNCS 763 (1993)