Safety Analysis in Formal Specification

Kaisa Sere and Elena Troubitsyna

Department of Computer Science, Åbo Akademi University,
Turku Centre for Computer Science (TUCS),
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland,
{Kaisa.Sere, Elena.Troubitsyna}@abo.fi

Abstract. Formal methods give us techniques to specify the functionality of a system, to verify its correctness or to develop the system stepwise from an abstract specification to its implementation. These aspects are important when designing safety-critical systems. Safety analysis is a vital part of the development of such systems. However, formal methods seldom interface well with the more informal techniques developed for safety analysis. Action systems is a formal approach to distributed computing that has proven its worth in the design of safety-critical systems. The approach is based on a firm mathematical foundation within which the reasoning about the correctness and behaviour of the system under development is carried out. The purpose of this paper is to show how we can incorporate the results of safety analysis into an action system specification by encoding this information via available composition operators for action systems in order to specify robust and safe controllers.

1 Introduction

Formal methods give us techniques to formally specify the functionality of a system, to verify its correctness or to develop the system stepwise from an abstract specification to its implementation. These aspects are important when designing safety-critical systems. Safety analysis is a vital part of the development of such systems. However, formal methods seldom interface well with the more informal techniques developed for safety analysis [13, 6]. Hansen et al. [5] spotted the problem of the semantic gap between the abstract level of the hazard analysis and the way of software specification. They suggest to use the results of Fault Tree Analysis as a source of the formulation of requirements which embedded software should meet. In their approach a description of fault trees is given in terms of real-time temporal logic. Their goal is to obtain a safety invariant which embedded software should preserve. Wong and Joyce [16] show how safety-related hazards are expressed in terms of source code for embedded software in order to verify this with respect to the hazards. The purpose of this paper is to develop a theory on how safety analysis techniques are used hand-in-hand with formal specification methods and how the results of the analysis are stepwise adopted by the specification in order to produce safe and robust systems consisting of both hardware and software.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1564–1583, 1999.

We use the action system formalism [1] as our formal design technique. This formalism is a state-based approach to system design. It provides a completely rigorous foundation for the stepwise development of system specifications and implementations. It has found many applications especially among parallel and distributed systems among which many are safety-critical [3, 10]. Our target systems are reactive, i.e., usually concurrent systems that interact with their environment and respond to not only normal safe situations, but to the occurred hazardous situations as well. We often call this environment a plant. Examples of such systems are embedded control systems.

In our earlier work [9, 14] we have proposed methods to reason about the impact of probabilistic behaviour of components on the overall safety of a control system. We concentrated on the probabilistic extension of the specification language developing tools to reason quantitatively about the systems' reliability and safety. Here we use the available modularization operators, most notably the prioritising composition [11], to capture the idea of safety related hazards. Our preliminary work [12] shows that the approach seems promising. We show in this paper how to embed the results of a hazard analysis into an action systems specification in a stepwise manner. The embedding is carried out within a formal calculus, the refinement calculus for action systems [1]. Our approach is similar to that of Hansen et al. in the sense that we can also obtain a safety invariant as a result of the safety analysis as they do. In addition, our approach allows the identified hazards to be specified and handled by the controller software. Hence, we focus on the specification aspect here.

Overview. In Section 2, we briefly describe action systems concentrating on the language issues and refinement as well as defining the important composition operators. In Section 3, we outline the way control systems are specified in the action systems framework. In Section 4, we show how the results of the Fault Tree Analysis can be encoded into the formalism. We exemplify the approach in Section 5. In Section 6, we concentrate on hazard analysis in a more general setting. We end in Section 7 with some concluding remarks.

2 Action Systems

An action system $\mathcal{A}$ is a set of actions operating on local and global variables:

$$\mathcal{A} \mathrel{\widehat{=}} \textbf{const}\ c; \textbf{global}\ z;\ |[\ \textbf{var}\ a\ ;\ A_0\ ;\ \textbf{do}\ A_1 \,[\!]\, \ldots \,[\!]\, A_n\ \textbf{od}\ ]|$$

The system $\mathcal{A}$ describes a computation, in which local variables a are first created and initialised in A_0. Then repeatedly any of the enabled actions $A_1, \ldots, A_n$ is non-deterministically selected for execution. The computation terminates if no action is enabled, otherwise it continues infinitely. Actions operating on disjoint sets of variables can be executed in any order or in parallel.

Actions are taken to be *atomic*, meaning that only their input-output behaviour is of interest. They can be arbitrary sequential statements. Their behaviour can therefore be described by the weakest precondition predicate transformer of Dijkstra [4]: $wp(A, p)$ is the weakest precondition such that action A

terminates in a state satisfying predicate p. In addition to the statements considered by Dijkstra, we allow pure guarded commands $g \rightarrow A$, non-deterministic choice $A \,[\!]\, B$ between actions A, B, and nondeterministic assignment $v := v'.Q$ which assigns to variables v such a value v' that the predicate Q holds.

$$\begin{array}{ll} wp(abort, p) \mathrel{\widehat{=}} false & wp((A\ ;\ B), p) \mathrel{\widehat{=}} wp(A, wp(B, p)) \\ wp(skip, p) \mathrel{\widehat{=}} p & wp((A \,[\!]\, B), p) \mathrel{\widehat{=}} wp(A, p) \wedge wp(B, p) \\ wp(v := e, p) \mathrel{\widehat{=}} p[v := e] & wp((g \rightarrow A), p) \mathrel{\widehat{=}} g \Rightarrow wp(A, p) \\ & wp(v := v'.Q, p) \mathrel{\widehat{=}} (\forall v'.Q \Rightarrow p[v := v']) \end{array}$$

Generally, an action that establishes any postcondition is said to be miraculous. We take the view that an action is only enabled in those initial states in which it behaves non-miraculously. The guard of an action characterises those states for which the action is enabled:

$$gd\ A \mathrel{\widehat{=}} \neg\, wp(A, false)$$

The body S of an action $A = g \rightarrow S$ is denoted by sA.

Let A and B be actions. The prioritising composition $A \,//\, B$ selects the first operand if it is enabled, otherwise the second, the choice being deterministic.

$$A \,//\, B \mathrel{\widehat{=}} A \,[\!]\, (\neg gd\ A \rightarrow B)$$

The prioritising composition of two actions is enabled if either operand is.

$$gd(A \,//\, B) = gd\ A \vee gd\ B$$

Let us now study different notions of refinement for action systems [2]. We say that action A is refined by action C, written $A \leq C$, if, whenever A establishes a certain postcondition, so does C:

$$A \leq C \text{ iff for all } p\text{: } wp(A, p) \Rightarrow wp(C, p)$$

Together with the monotonicity of wp this implies that for a certain precondition, C might establish a stronger postcondition than A (reduce nondeterminism of A) or even establish postcondition *false* (behave miraculously).

A variation of refinement is if A is (data-) refined by C via a relation R, written $A \leq_R C$. For this, assume A operates on variables a, u and C operates on variables c, u. Let R be a predicate over a, c, u:

$$A \leq_R C \ \text{ iff for all } p\text{: } R \wedge wp(A, p) \Rightarrow wp(C, (\exists\, a \cdot R \wedge p))$$

Data refinement allows the local variables of an action system to be replaced. We have the following theorem to prove data refinement between actions:

Theorem 1. *$A \leq_R C$ holds iff*

(i) $R \wedge gd\ C \Rightarrow gd\ A$
(ii) for all p: $R \wedge gd\ C \ \wedge \ wp(sA, p) \Rightarrow wp(sC, (\exists a \cdot R \wedge p))$

Rule 1 $a{:=}? \leq a{:=}a'.Q$,where $a{:=}? \mathrel{\widehat{=}} a{:=}a'.true$
Rule 2 For two actions A, B: $A \parallel B \leq A \mathbin{//} B$
Rule 3 $g1 \vee g2 \rightarrow abort \mathbin{//} g(A) \rightarrow A \leq g1 \rightarrow abort \mathbin{//} g2 \rightarrow B \parallel g(A) \rightarrow A$ where A and B can also be abortive
Rule 4 $a{:=}? \leq_R c{:=}c'.Q$ if $Q \wedge R \Rightarrow (\exists a' \cdot R[a, c{:=}a', c'])$
Rule 5 $A1;A2 \leq_R C1;C2$ if $A1 \leq_R C1$ and $A2 \leq_R C2$
Rule 6 $A1 \mathbin{//} A2 \leq_R C1 \mathbin{//} C2$ if $A1 \leq_R C1$ and $A2 \leq_R C2$ and $R \wedge gd\, A1 \Rightarrow gd\, C1$

Fig. 1. Refinement rules

The next theorem presents conditions to be verified in order to establish refinement between action systems.

Theorem 2. *$\mathcal{A} \leq_R \mathcal{C}$ holds iff*

(i) $C0 \Rightarrow (\exists a \cdot R \wedge A0)$
(ii) $A \leq_R C$
(iii) $R \wedge gd\, A \Rightarrow gC$

The proofs of Theorem 1 and Theorem 2 can be found elsewhere [2].

When carrying out refinement in practice one seldom appeals to the general definition of refinement. Instead certain pre-proven refinement rules are used. Figure 1 presents a number of rules [12] that are especially useful when working with hazards as will be seen later.

3 Specifying Control Systems with Safety Consideration

Let us now sketch a way to specify control systems within action systems formalism. Rather than embody all the requirements in the initial specification, we introduce some of them in successive refinement steps. Usually, refinement is used as a way of verifying the correctness of an implementation with respect to a specification, but is it can also be used as a way of structuring the requirements such that they are easier to validate [3, 10]. In this paper we develop mechanisms to handle failure situations by the refinement activity.

Our initial action system is intended to model the behaviour of the overall system, that is, the physical environment and the controller together. It allows us to use assumptions that we make about how the environment behaves. The initial specification of the system is very abstract. Usually it is built in such a way that all the details concerning interaction between the plant and the controller (via sensors and actuators) as well as details of failures are omitted.

Below a control system is modelled as an interleaving between the plant P and the controller C

$$System \mathrel{\widehat{=}} \textbf{const}\ c;\ \textbf{global}\ z;\ |[\ \textbf{var}\ pv, cv, fail;\ I;\ \textbf{do}\ P\ ;\ C\ \textbf{od}\ ]| \qquad (1)$$

The action I initialises the system. Both P and C are actions and they might share variables. This initial specification of the plant action P is

$$P \mathrel{\widehat{=}} pv, z, fail: =?, ?, ?$$

where pv are the state variables needed to model the local state of the plant, and the controller action C is

$$C \mathrel{\widehat{=}} Failure \mathbin{//} (Unit_1 \parallel Unit_2 \parallel \ldots \parallel Unit_M)$$

The controller consists of a prioritising composition of the action *Failure*

$$Failure \mathrel{\widehat{=}} fail \rightarrow Emergency$$

which shuts down the system if a failure occurs, i.e., the action *Emergency* is equivalent to *abort*, and the actions $Unit_i$, $i = 1..M$, which have the form

$$Unit_i \mathrel{\widehat{=}} g_i \rightarrow control\ action_i$$

Here for simplicity we assume that the occurrence of a global system failure is modelled by a local variable *fail*. Later we drop this simplification and consider the action *Failure* guarded by a predicate over the global and the local variables of the system. Each of the actions $Unit_i$ specifies the control required to operate a certain plant device (we call it a plant unit) in absence of failures. They can refer to the variables pv, z, and normally some other variables, too, denoted as cv in (1) . Observe that we can be certain that there are no failures present when an action $Unit_i$ is executed as the prioritising composition between the faulty behaviour and the control actions ensures this. Without this operator, i.e., using the choice operator between these actions, the control actions would require a more elaborate guard, namely $g_i \wedge \neg fail$. This latter approach is taken for instance by Liu and Joseph [8].

In our initial specification *System* we assume that the state of the plant can be directly observed by the controller. Further refinement of the initial specification leads to the introduction of implementation details which make the specification more realistic: the controller cannot observe the real state of the plant any more but rather makes assumptions about it based on sensor readings. Control is performed by means of actuators which, like the sensors, are modelled as state variables [3]. Eventually, we arrive at the representation of the system in the form presented in Fig. 2.

In our previous work on action systems for safety-critical systems [3, 10], failure modes of the components together with the safety invariant imposed on the system were given a priori. The task was to capture these requirements into a specification. In the industrial practice, however, the design of a safety-critical system assumes that this information is unavailable and should be obtained as a result of a safety analysis of the system. On the base of the safety analysis the designer should build the controller so that it is able to withstand faults appearing in the fault-prone units. To obtain the failure modes for the controlling program we show how the specification and refinement of the system under construction can proceed hand-in-hand with the safety analysis.

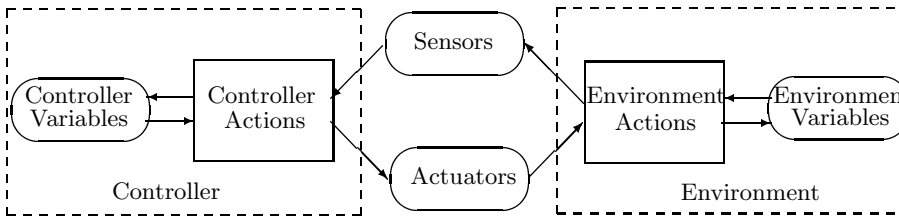

Fig. 2. Structure of the system specification

Observe, that both the safety analysis and the stepwise program development conduct reasoning from an abstract level to a concrete level. The safety analysis starts by identifying hazards which are potentially dangerous, and proceeds by producing detailed descriptions of them together with finding the means to cope with these hazards. We incorporate the information that becomes available at each stage of the safety process by performing corresponding refinements of the initial specification, as shown in Fig. 3.

With such an approach the preliminary hazard identification gives the semantics to the action *Failure*: it is a reaction on the occurrence of the identified hazard. When the system enters a hazardous state it violates the safety invariant and therefore, should be shut down. Safe operation of the action system *System* (1) can be expressed via a safety invariant *safety* on the state variables of the system:

$$safety \mathrel{\widehat{=}} \neg fail \;\Rightarrow\; safety\ condition$$

Safety is checked within the weakest precondition calculus by ensuring that the initial state establishes *safety*, and that $wp(C, safety)$ holds, i.e. the actions of the controller preserve *safety*. As the safety analysis proceeds more information on the failures causing hazardous situations becomes available and allows us to weaken the safety invariant by expressing this new information.

4 Representing a Fault Tree in a Specification

There are a number of standard techniques for producing detailed description of the identified hazards [6, 13]. In this paper we choose the Fault Tree Analysis and show next, how to incorporate the information obtained as a result of the fault tree analysis in the initial system specification (1) given in form of an action system. The Fault Tree Analysis (FTA) is a deductive safety analysis technique. It is a top-down approach applied during the entire design stage. A preliminary hazard identification provides information about functions of the system and the possible failures (see Fig. 3). This information is taken as an input for the FTA. The result of the FTA is an identification of those component faults that result in different hazardous situations. Each fault tree has a root representing

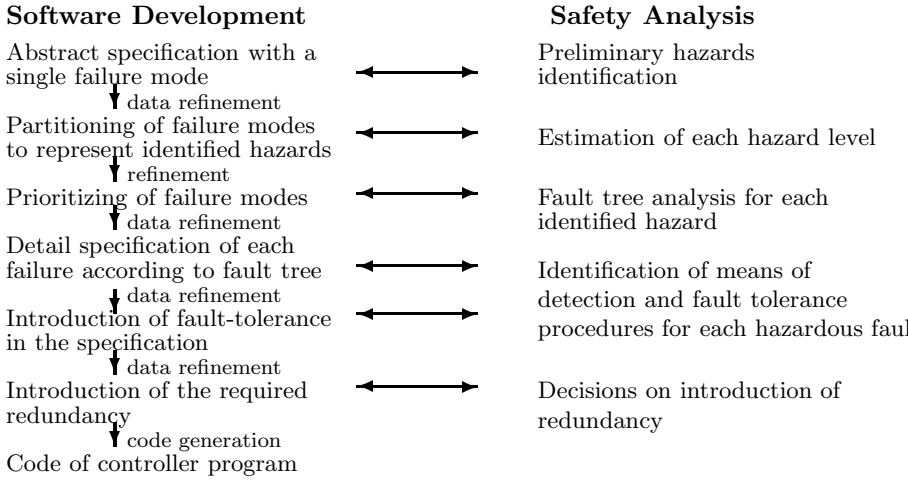

Fig. 3. Interaction of software development process with safety analysis

a hazardous failure. The tree traces the system to the lower component level revealing the possible causes of the failure.

A fault tree consists of two main entities, leaves and gates. The leaves, often called events, represent the system states which in combination with other events lead to the occurrence of the hazardous fault represented by the root of the tree. The gates are logical connectives which join the leaves. They describe which particular combination of events results in the occurrence of the hazard. In this paper we consider two basic logical gates, namely disjunction and conjunction. Below we present the rules which allow us to embed the information about failures given in the form of a fault tree in the specification of the system given in the form of an action system.

A leaf of a fault tree describes a certain set of system states. It, therefore, can be expressed as a predicate over the state variables of the system. In the system specification the leaves, or more precisely the predicates representing the leaves, appear as the guards of the actions which specify the reaction of the system on the occurred faults. With each event we also associate a certain level of criticality defined by the level of the fault tree at which that particular event appears. The root of the tree, therefore, is the event of the first level of criticality, the events directly connected to the root by means of a logical gate have the second level of criticality etc. It is clear, that the occurrence of an event which is close to the root might lead to an inevitable catastrophe, and therefore, should be dealt with urgently.

A gate of a fault tree defines the logical operator (conjunction or disjunction) over the predicates representing the leaves which the gate connects. Therefore, if an action specifies a reaction of the controller on the combination of certain events we define its guard on the base of the gate which conjoins these events.

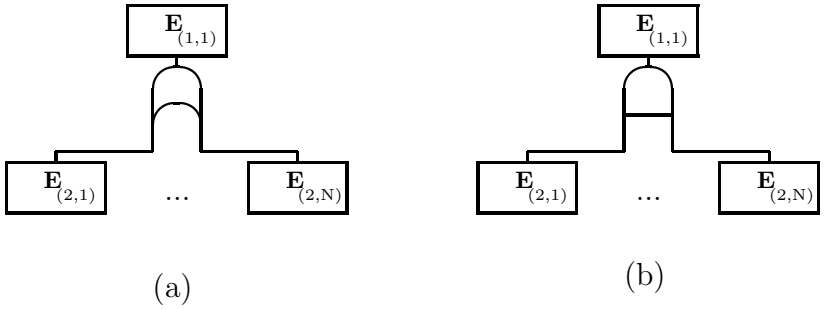

Fig. 4. Basic fault trees

The approach we advocate in this paper suggests to analyse a fault tree in a stepwise manner. Namely, we start from the specification of the system in form (1) where the value *true* of the boolean *fail* represents the root of the fault tree. Here for simplicity, we assume that there is only one hazard identified for a given system. In the next section we extend the technique to reason about systems with several hazards. Analysing the fault tree level after level we stepwise embed detailed representation of the faults and model the reaction of the controller in the specification. As a result, we obtain a specification of the system within which both faults and reactions on them are specified in terms of the state variables. Moreover, the faults are treated according to their criticality which is defined by the levels of the fault tree. Below we describe a number of generic rules which allow us to specify faults preserving the structure of the fault tree.

Consider the fault tree (a) in Fig. 4, where events $E_{(2,1)}, \ldots, E_{(2,N)}$ are caused by failures of sensors and actuators (see Fig.2) or represent certain events over globally observed system states (e.g. the states of the physical environment). The event $E_{(i,j)}$ stands for the $j-th$ event on the $i-th$ level of the fault tree. Even though we consider here only two levels, the results can be applied recursively to an arbitrary number of levels. We show an example of this in the next section.

The occurrence of the failure $E_{(1,1)}$ and the system reaction on that can be specified by the action *Failure* of the following form:

$$Failure \mathrel{\widehat{=}} E_{(2,1)} \vee \ldots \vee E_{(2,N)} \rightarrow \ Reaction\ on\ E_{(1,1)}$$

where $E_{(2,1)}, \ldots, E_{(2,i)}$ are predicates over the local variables of the system representing sensor and actuator failures and $E_{(2,i+1)}, \ldots, E_{(2,N)}$ are predicates over the global variables representing events over globally observed system states. Moreover,

$$E_{(1,1)} = E_{(2,1)} \vee \ldots \vee E_{(2,N)}$$

Data refinement allows us to change the local part of the state space (i.e. to manipulate the local variables of the specification) provided the behaviour of

the system on the global level is preserved. Therefore, the events expressed as predicates over the global variables of the system should obtain a detail representation already in the initial specification of the system as new globally observable behaviour cannot be added via refinement. This restriction allows us to ensure that the behaviour of the refined specification is subsumed by the behaviour of the initial specification. In contrary, the representation of the events which do not refer to the global state can be very abstract in the initial specification.

Consider again the fault tree (a) in Fig. 4 where none of the events $E_{(2,1)}$, ..., $E_{(2,N)}$ refers to the global state and hence are for simplicity modelled by the local variables $E_{(2,1)}, \ldots, E_{(2,N)}$ in the system specification. For that case we have the following result:

Theorem 3. *The action system $\mathcal{A}$ of the form (1) such that $E_{(1,1)} = fail$ in the fault tree (a) in Fig. 4 is refined by the action system $\mathcal{A}'$:*

$$\mathcal{A}' \mathrel{\widehat{=}} \textbf{const } c; \textbf{ global } z; \ [\![\ \textbf{var } v, E_{(2,1)}, ..., E_{(2,N)}; \ I'; \ \textbf{do } P \ ; \ C' \textbf{ od } \ ]\!]$$

where I' initialises the variables and the controller action C' is a prioritising composition of the reaction on the occurred failure $E_{(1,1)}$ specified by the action Failure′

$$Failure' \mathrel{\widehat{=}} E_{(2,1)} \vee \ldots \vee E_{(2,N)} \rightarrow Emergency$$

with the control actions:

$$C' \mathrel{\widehat{=}} Failure' \ // \ (Unit_1 \ [\!] \ Unit_2 \ [\!] \ \ldots \ [\!] \ Unit_M)$$

Proof. The refinement relation $R \mathrel{\widehat{=}} fail = (E_{(2,1)} \vee \ldots \vee E_{(2,N)})$ allows us to prove that $\mathcal{A} \leq_R \mathcal{A}'$ appealing to Theorem 2, hence, proving the theorem.

Next we develop a similar rule for specifying the fault tree (b) in Fig. 4 which contains the logical gate conjunction. Assume that events $E_{(2,1)}, \ldots, E_{(2,i)}$ are caused by failures of sensors and actuators (see Fig.2) and events $E_{(2,i+1)}, \ldots, E_{(2,N)}$ represent certain events over globally observed system states. The occurrence of the failure $E_{(1,1)}$ is caused by the conjunction of these events as specified by the fault tree. We specify the occurrence of the event $E_{(1,1)}$ and the reaction of the controller on that by the action *Failure* of the following form:

$$Failure \mathrel{\widehat{=}} E_{(2,1)} \wedge \ldots \wedge E_{(2,N)} \rightarrow \ Reaction \ on \ E_{(1,1)}$$

where $E_{(2,1)}, \ldots, E_{(2,i)}$ are again predicates over the local variables of the system and $E_{(2,i+1)}, \ldots, E_{(2,N)}$ are predicates over the global variables and

$$E_{(1,1)} = E_{(2,1)} \wedge \ldots \wedge E_{(2,N)}$$

Note, that in case a failure is caused by disjunction of a set of events (the fault tree (a) in Fig. 4) the controller is intolerant to the occurrence of any single event from this set. In case of conjunction (the fault tree (b) in Fig. 4) the situation is different: the controller can cope with each particular event to

preclude the occurrence of the more critical failure caused by the conjunction of these events.

Again for simplicity assume that none of the events $E_{(2,1)}, \ldots, E_{(2,N)}$ of the fault tree (b) in Fig. 4 refers to the global states and are therefore modelled by local variables of the same name in the system specification. Then the following theorem provides us with a formal technique to represent such a fault tree in the specification as a refinement of the initial system specification.

Theorem 4. *The action system $\mathcal{A}$ of the form (1) such that $E_{(1,1)} = fail$ in the fault tree (b) in Fig. 4 is refined by the action system $\mathcal{A}'$:*

$$\mathcal{A}' \mathrel{\widehat{=}} \textbf{const } c; \textbf{ global } z; \ [\![\ \textbf{var } v, E_{(2,1)}, \ldots, E_{(2,N)}; \ I'; \ \textbf{do } P \ ; \ C' \textbf{ od }]\!]$$

where I' is the new initialisation and the controller action C'

$$C' \mathrel{\widehat{=}} Failure' \ // \ (Unit_1 \ [\!] \ Unit_2 \ [\!] \ \ldots \ [\!] \ Unit_M)$$

is a prioritising composition between the control actions $Unit_1$, $Unit_2$, $\ldots$, $Unit_M$ and the action $Failure'$

$$\begin{aligned} Failure' \mathrel{\widehat{=}} \ & E_{(2,1)} \wedge \ldots \wedge E_{(2,N)} \rightarrow Emergency \\ & // \ E_{(2,1)} \rightarrow Rescue_{E_{(2,1)}} \\ & [\!] \ \ldots \\ & [\!] \ E_{(2,N)} \rightarrow Rescue_{E_{(2,N)}} \end{aligned}$$

which specify the reaction Emergency of the controller on the occurrence of the hazardous failure $E_{(1,1)}$, together with the reaction statements $Rescue_{E_{(2,1)}}, \ldots, Rescue_{E_{(2,N)}}$ on the local variables $E_{(2,1)}, ..., E_{(2,N)}$.

Proof. The refinement relation $R \mathrel{\widehat{=}} fail = (E_{(2,1)} \vee \ldots \vee E_{(2,N)})$ allows us to prove that $\mathcal{A} \leq_R \mathcal{A}'$ on the base of Theorem 2. This results in proving the theorem.

The statements $Rescue_{E_{(2,1)}} \ \ldots, Rescue_{E_{(2,N)}}$ specify invocations of the maintenance procedures as the responses on the occurred failures of the sensors and the actuators (see Fig.2).

The treatment of a more general case where the events $E_{(2,1)}, \ldots, E_{(2,N)}$ can refer to both global and local states is different in the sense that we have to give a detailed description of the events over the global system state already in the initial specification. The reasoning about the events referring to the local part of the state space is, however, still conducted as above.

Below we present a general form of the initial specification of the action *Failure* for this case which additionally specifies controller reactions on combined events caused by multiple failures. In that case the action representing the occurrence of failure $E_{(1,1)}$ of the fault tree (b) in Fig. 4 contains also the actions specifying the reactions on these combined events. The specification of the occurrence of the failure $E_{(1,1)}$ as well as the occurrences of the combinations of the events $E_{(2,1)}, \ldots, E_{(2,N)}$ with the reactions of the controller are represented by the action *Failure* below

$$
\begin{aligned}
\mathit{Failure} \mathrel{\widehat{=}} & \; E_{(2,1)} \wedge \ldots \wedge E_{(2,N)} \rightarrow \mathit{Emergency} \\
// & \bigwedge_{i \in [1..N]} E_{(2,i)} \rightarrow \mathit{Resque}_1 \\
[\!] & \ldots \\
[\!] & \bigwedge_{i \in [1..N]} E_{(2,i)} \rightarrow \mathit{Resque}_l
\end{aligned}
\tag{2}
$$

where $E_{(2,1)}, \ldots, E_{(2,N)}$ are predicates over the local and the global variables. Here the statements $\mathit{Resque}_1, \ldots, \mathit{Resque}_l$ specify the invocations of the maintenance procedures as the responses on the occurred events. The guards of these actions are formed from arbitrary event combinations and might describe reactions on each of the events $E_{(2,1)}, \ldots, E_{(2,N)}$ separately as well.

5 Example: A Heater Controller

To illustrate both construction of a fault tree and building of the corresponding specification we consider an example — a heater controller for a tank of toxic liquid. A computer controls the heater using a power switch on the basis of information obtained from a temperature sensor. The controller tries to maintain the temperature between certain limits. If the temperature exceeds a critical threshold the toxic liquid can harm its environment in a certain way (we leave it unspecified).

We start the safety analysis of our system (see Fig. 3) by the preliminary hazard identification. Since the system can harm its environment as a result of overheating of the toxic liquid, we identify the hazard *overheating* and proceed the analysis by constructing the corresponding fault tree. The fault tree in Fig. 5 identifies the faults of the system components and their logical combinations which lead to overheating.

Overheating of the toxic liquid, the event $E_{(1,1)}$ takes place if the temperature reaches a predefined threshold, heat is supplied, and a failure to switch off the heater takes place. Therefore, $E_{(1,1)} = E_{(2,1)} \wedge E_{(2,2)} \wedge E_{(2,3)}$. The failure to switch of the heater, the event $E_{(2,3)}$ is a result of the failure to issue the switch off signal or a primary switch failure, $E_{(2,3)} = E_{(3,1)} \vee E_{(3,2)}$. Finally, the failure $E_{(3,1)}$ occurs if either the controller fails or the temperature sensor fails and indicates a wrong (lower than the real) temperature, $E_{(3,1)} = E_{(4,1)} \vee E_{(4,2)}$.

Designing a formal specification of the system according to the approach proposed in this paper, we depict the information obtained from the construction of the fault tree. Our initial specification of the system below:

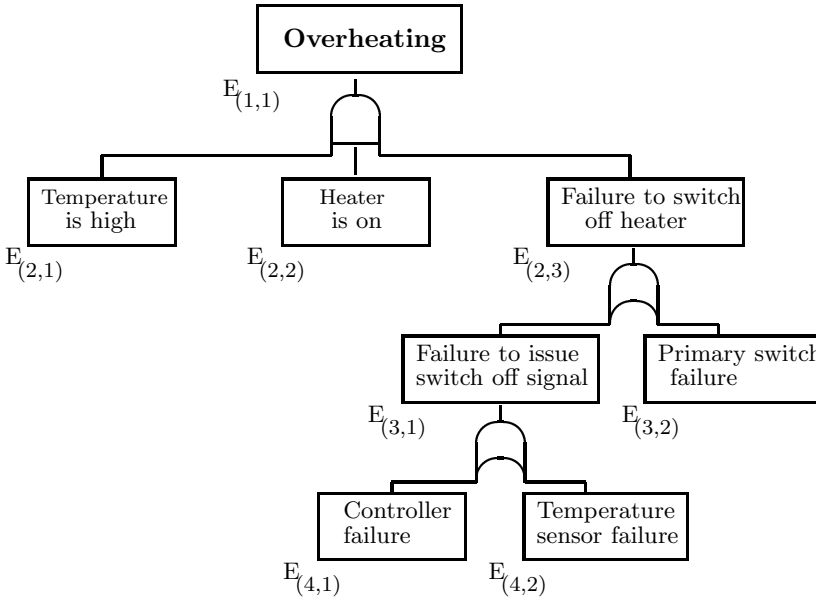

Fig. 5. Fault tree of overheating

$\mathcal{A} \mathrel{\widehat{=}}$ **const** *tr*: *Real* / ∗ *critical threshold* ∗ /
ht: *Real* / ∗ *high temperature limit* ∗ /
lt: *Real* / ∗ *low temperature limit* ∗ /
maxd: *Real* / ∗ *maximal temperature decrease per unit of time* ∗ /
maxi: *Real* / ∗ *maximal temperature increase per unit of time* ∗ /
maxt: *Real* / ∗ *maximal feasible temperature* ∗ /
$lt < ht < tr$;
global *t*: *Real* / ∗ *temperature* ∗ /
heat: *on*|*off* / ∗ *supply of heating* ∗ /
|[**var** $E_{(2,3)}$:*Bool* ; *I* ; **do** *Environment* ; *Controller* **od**]|

has the form (1) where *Controller* = *Failure* // *Switch*.

As described previously, the action *Failure* specifies the occurrence of overheating and the rescue procedures undertaken by the controller as responses to the occurred failures. Overheating is the result of the conjunction of the events $E_{(2,1)}$, $E_{(2,2)}$, $E_{(2,3)}$. We express the event $E_{(2,1)}$ by the predicate $t \geq tr$, where t is a global variable modelling the temperature of the liquid and tr is the critical temperature threshold defined by the corresponding constant in our specification. Similarly, the event $E_{(2,2)}$ is represented by the predicate $heat = on$. The global variable *heat* evaluates to *on* if the heater is switched on and to *off* otherwise. The event $E_{(2,3)}$ is caused by the failures of the sensor and the actuator,

which will appear in the specification later. Meanwhile we model the failure $E_{(2,3)}$ by a local variable with the same name: the variable $E_{(2,3)}$ is *true* if the event $E_{(2,3)}$ takes places. Therefore, we define overheating, the event $E_{(1,1)}$, as follows:

$$overheating = t \geq tr \wedge heat = on \wedge E_{(2,3)}$$

If overheating takes place the system should be shut down. However, if no failure to switch off heating occurs the controller can preclude an immediate occurrence of overheating by switching off the heater. We pessimistically assume that the system is shut down if there is a failure to switch off the heater. We specify a more realistic treatment of this failure as soon as a detailed specification of the sensor and the actuator becomes available. The specification of the action *Failure*

$$\begin{aligned} Failure \mathrel{\widehat{=}}\ & overheating \rightarrow abort \\ [\!] \ & t \geq tr \wedge heat = on \rightarrow heat := off \\ [\!] \ & E_{(2,3)} \rightarrow abort \end{aligned}$$

is obtained on the basis of the reasoning described in Section 4. Observe, that the structure of the part of the fault tree in Fig. 5 we analyse here is similar to the structure of the fault tree (*b*) in Fig. 4. The general form of an action specifying the responses of the system to combined events was given by the action (2). Here we applied the same kind of reasoning to obtain the action *Failure*.

The rest of the system specification is rather typical for control systems treated within the action system formalism. Specifying the initialisation we assume that the system starts its operation in a state where no failures have occurred: $I \mathrel{\widehat{=}} E_{(2,3)} := false$.

We specify the environment very abstractly: we merely describe an arbitrary temperature change and a non-deterministic occurrence of a fault.

$$\begin{aligned} Environment &= Env_p; Env_f \\ Env_p &= t := ? \\ Env_f &= E_{(2,3)} := ? \end{aligned}$$

The control action *Switch* specifies the switching off and on the heater in order to maintain the liquid temperature in the safe region:

$$\begin{aligned} Switch \mathrel{\widehat{=}}\ & t \geq ht \wedge heat = on \rightarrow heat := off \\ [\!] \ & t \leq lt \wedge heat = off \rightarrow heat := on \end{aligned}$$

The analysis of the next level of the fault tree is based on the application of Theorem 3: the occurrence of the event $E_{(2,3)}$ results from the disjunction of the events $E_{(3,1)}$ and $E_{(3,2)}$. To specify the event $E_{(3,1)}$ we introduce a local variable $E_{(3,1)}$ which is *true* if the event occurs. The event $E_{(3,2)}$ results from the failure of the actuator — the power switch. To specify this we introduce the local variable *sw_stat* modelling the status of the switch in our specification. To simplify the reasoning we omit the detailed specification of an invocation of a

switch repair procedure and present only its effect: the repaired power switch. The specification of the response on the event $E_{(3,1)}$ is similar to that of the event $E_{(2,3)}$.

The specification of the system $\mathcal{A}'$ is as follows

$$\begin{aligned}\mathcal{A}' \mathrel{\widehat{=}} \ldots \ [\![\ & \textbf{var } \mathit{sw\_stat}{:}\mathit{ok}|\mathit{failed} \ ; \ E_{(3,1)}{:}\mathit{Bool} \ ; \ I' ; \\ & \textbf{do } \mathit{Environment}'; \mathit{Controller}' \ \textbf{od} \ ]\!]\end{aligned}$$

where $\mathit{Controller}' = \mathit{Failure}' \ // \ \mathit{Switch}$ is obtained by taking into account information obtained from the analysis of the second and the third levels of the fault tree in Fig. 5 as specified by the action $\mathit{Failure}'$

$$\begin{aligned}\mathit{Failure}' \mathrel{\widehat{=}} \ & \mathit{overheating}' \rightarrow \mathit{abort} \\ & // \ t \geq \mathit{tr} \wedge \mathit{heat} = \mathit{on} \rightarrow \mathit{heat}{:=}\mathit{off} \\ & // \ \mathit{sw\_stat} = \mathit{failed} \rightarrow \mathit{sw\_stat}{:=}\mathit{ok} \\ & [\!] \ E_{(3,1)} \rightarrow \mathit{abort}\end{aligned}$$

Here $\mathit{overheating}' = t \geq \mathit{tr} \wedge \mathit{heat} = \mathit{on} \wedge (\mathit{sw\_stat} = \mathit{failed} \vee E_{(3,1)})$.

Also in this step we refine the environment action by considering maximal system dynamics and by modelling the occurred failures over the introduced local variables:

$$\begin{aligned}\mathit{Environment}' &\mathrel{\widehat{=}} \mathit{Env}'_p; \mathit{Env}'_f \\ \mathit{Env}'_p &\mathrel{\widehat{=}} t{:=}t'.t - \mathit{maxd} \leq t' \leq t + \mathit{maxi} \wedge 0 \leq t \leq \mathit{maxt} \\ \mathit{Env}_f &\mathrel{\widehat{=}} \mathit{sw\_stat}{:=}? \ ; \ E_{(3,1)}{:=}?\end{aligned}$$

The new initialisation is $I' \mathrel{\widehat{=}} \mathit{sw\_stat}{:=}\mathit{ok} \ ; \ E_{(3,1)}{:=}\mathit{false}$

On the base of Theorem 3 and the refinement rules given in Fig.1 it can be shown that the action system $\mathcal{A}'$ refines the action system $\mathcal{A}$ with the refinement relation R_1

$$R_1 \mathrel{\widehat{=}} E_{(2,3)} = (\mathit{sw\_stat} = \mathit{failed} \vee E_{(3,1)})$$

Analysing the last level of the fault tree in Fig. 5, we observe that the event $E_{(4,1)}$ cannot be expressed in the specification of the controller. The failure of the controller is caused by the hardware or software error. However, it points out the necessity to introduce a controller independent device in the system design, a watch dog. Such a device periodically checks the status of the controller and shuts down the system or activates a stand-by controller if the main controller fails. Therefore, we consider the event $E_{(4,2)}$ which specifies a failure of the temperature sensor. The introduction of a representation of the sensor in the system specification transforms the specification of the controller in such a way that the controller relies on the sensor readings to perform its duties. The real state of the environment becomes inaccessible to the controller. Applying Theorem 3 we perform data refinement of the system obtaining the specification $\mathcal{A}''$.

$$\begin{aligned}\mathcal{A}'' \mathrel{\widehat{=}} \ldots \ [\![\ & \textbf{var } \mathit{sw\_stat}, \mathit{sen\_stat}{:}\mathit{ok}|\mathit{failed} \ ; \ t\_tr, t\_est_1, t\_est_2{:}\mathit{Real} \ ; \ I'' ; \\ & \textbf{do } \mathit{Environment}''; \mathit{Controller}'' \ \textbf{od} \ ]\!]\end{aligned}$$

where $Controller'' = Failure'' \;//\; Switch'$. The initialisation establishes a state where both the temperature sensor and the power switch function properly.

$$I'' \mathrel{\widehat{=}} sw\_stat := ok \;;\; sen\_stat := ok; t\_tr := t; t\_est_1, t\_est_2 := t\_tr, t\_tr$$

The environment models a change of the temperature, independent occurrences of the sensor and the actuator failures, and an estimate of the temperature made by the controller:

$$\begin{array}{lll} Environment'' & \mathrel{\widehat{=}} & Env_p' \;;\; Env_f'' \;;\; T\_Estim \\ Env_f'' & \mathrel{\widehat{=}} & sw\_stat :=? \;;\; sen\_stat :=?; \\ & & \quad t\_tr := t\_tr'.sen\_stat = ok \Rightarrow t\_tr' = t \wedge \\ & & \qquad\qquad sen\_stat = failed \Rightarrow t\_tr' = t\_tr \\ T\_Estim & \mathrel{\widehat{=}} & t\_est_1, t\_est_2 := t\_est_1', t\_est_2'.Q \end{array}$$

where

$$\begin{array}{l} Q = (sen\_stat = ok \Rightarrow t\_est_1' = t\_tr \wedge t\_est_2' = t\_tr) \wedge (sen\_stat = failed \Rightarrow \\ \quad (t\_est_1' = (if\ t\_est_1 + maxi < maxt\ then\ t\_est_1 + maxi\ else\ maxt) \wedge \\ \quad t\_est_2' = (if\ t\_est_2 - maxd > 0\ then\ t\_est_2 - maxd\ else\ 0))) \end{array}$$

Compared to the action $Failure'$, the action $Failure''$ introduces additionally a reaction to the failure of the sensor. Moreover, it defines overheating by tracing the whole fault tree (Fig. 5):

$$\begin{array}{lll} Failure'' = & overheating'' \rightarrow abort \\ & // \; t\_est_1 \geq tr \wedge heat = on \rightarrow heat := off \\ & // \; sw\_stat = failed \rightarrow sw\_stat := ok \\ & // \; sen\_stat = failed \rightarrow sen\_stat := ok \end{array}$$

with $overheating'' = t\_est_1 \geq tr \wedge heat = on \wedge (sw\_stat = failed \vee sen\_stat = failed)$.

In the specification of the controller we change the access to the real temperature and substitute it by the temperature estimate of the controller:

$$\begin{array}{lll} Switch' & \mathrel{\widehat{=}} & t\_est_1 \geq ht \wedge heat = on \rightarrow heat := off \\ & [\!] & t\_est_2 \leq lt \wedge heat = off \rightarrow heat := on \end{array}$$

Data refinement between the action systems $\mathcal{A}'$ and $\mathcal{A}''$, $\mathcal{A}' \leq_{R2} \mathcal{A}''$, is proved with the refinement relation

$$\begin{array}{l} R2 \mathrel{\widehat{=}} E_{(3,1)} = sen\_stat \wedge \\ \quad (sen\_stat = ok \Rightarrow t = t\_tr \;\wedge t\_est_1 = t\_tr \wedge t\_est_2 = t\_tr) \wedge \\ \quad (sen\_stat = failed \Rightarrow t\_est_2 \leq t \leq t\_est_1) \end{array}$$

6 Prioritising Hazards

Section 4 provided us with techniques that allow us to represent a single hazard in a specification. Often, however, there are several hazards identified for a system under construction. We need, therefore, to generalise the presented approach to reasoning about system hazards in general.

We assume that a set of hazards $\mathcal{H}$ is obtained as a result of the hazard identification. For each hazard $H_i \in \mathcal{H}$ an appropriate fault tree FT_i is constructed. The obtained fault trees form a set $\mathcal{FT}$: each tree from the set can be represented in the system specification as described in Section 4. Here we focus on the interaction between the representation of hazards in the specification.

Analysing the set of hazards $\mathcal{H}$ we assess the risk associated with each hazard from this set. The assessment is based on available quantitative information about component reliabilities or on expert judgements about the likelihood and severity of each hazard. Having assessed the risks associated with the identified hazards we can classify them. There are a number of methods and standards providing guidance for the classification of risks [13]. Without going into details we assume without loss of generality that there are three disjoint classes of hazards formed on the basis of the classification of risks associated with the hazards.

$$\begin{aligned} &\textit{Class I}: && \{H_1, \ldots, H_{c_1}\} \\ &\textit{Class II}: && \{H_{c_1+1}, \ldots, H_{c_2}\} \\ &\textit{Class III}: && \{H_{c_2+1}, \ldots, H_{c_3}\} \end{aligned}$$

Let us make this more concrete by giving a potential interpretation to the classes. Assume that on the base of the performed classification we formulate failure modes of the system to be designed. The system enters *Emergency* mode if any of the hazardous situations from *Class I* occurred. Hence, these are the hazards that are intolerable and have a high risk associated with them. The mode *Resque* is caused by hazards belonging to class *Class II*. These are less severe hazards but still critical. They should be avoided or their effect should be mitigated. An occurrence of a hazard from *Class III* transforms the system into the *Degraded* mode. Here the failures can be tolerated as the risks associated with the corresponding hazards are negligible.

Now we return to the specification of the system from the software point of view. Developing the specification of the controller which should withstand several types of hazardous failures it is desirable to carry out the development process in such a way that the produced classification of hazards is preserved.

Consider again the general form of the system specification (1). As we described previously the guard of the action *Failure*, *fail* is in general a predicate over the global and the local variables. It expresses the occurrence of the identified hazardous failure. Since we now consider a set of hazardous failures the predicate *fail* should express the occurrence of any of them, i.e.

$$\textit{fail} = \bigvee_{i=1}^{N} H_i$$

where each of the predicates H_i, $H_i \in \mathcal{H}$ for $i = 1..N$ describes a corresponding hazard in terms of the state variables as explained in Section 4. In the initial system specification we assume pessimistically, that each of the hazardous faults is treated equally by *Emergency* statement.

To introduce the different failure modes in the general specification of failures given by the action *Failure* we partition it as shown below:

$$Failure = Fail1 \parallel Fail2 \parallel Fail3$$

The three actions $Fail1, Fail2, Fail3$ which describe the different classes of hazards:

$$Fail1 \mathrel{\widehat{=}} gFail1 \rightarrow Emergency$$
$$Fail2 \mathrel{\widehat{=}} gFail2 \rightarrow Resque$$
$$Fail3 \mathrel{\widehat{=}} gFail3 \rightarrow Degraded$$

The action *Fail*1 specifies the reaction of the system on hazards from *Class I* which is *Emergency*, shut down of the system. The occurrence of a hazard or several of them is modelled by the guard of the action, which is defined to be disjunction of hazards from *Class I*:

$$gFail1 \mathrel{\widehat{=}} \vee_{i=1}^{c_1} H_i$$

The hazards belonging to *Class II* are specified by the action *Fail*2. Since a hazard from *Class II* does not lead to the imminent catastrophe, some actions to bring the system back to a non-hazardous state should be undertaken. Generally, the action has the form

$$Fail2 \mathrel{\widehat{=}} H_{c_1+1} \rightarrow Resque_{c_1+1} \ \parallel \ldots \ \parallel H_{c_2} \rightarrow Resque_{c_2}$$

Here each of the individual actions becomes enabled if a corresponding hazard from *Class II* occurs. The body of each action is an invocation of some *Resque* procedure. The structure of the action *Fail*3 is similar to the action *Fail*2, but has the hazards of *Class III* as the guards and corresponding corrective procedures as the bodies.

Another safety requirement which we capture in the specification is a necessity to cope with the failures according to their criticality: we give priority to failures with high risks associated to them. Hence, *Fail*1 should be executed immediately when enabled. Also *Fail*2 and *Fail*3 will be taken whenever enabled provided no action in a higher priority class is enabled. A normal control action $Unit_i$ is only taken when there are no failures detected in the system. Therefore, the most severe hazards — hazards belonging to *Class I* should be handled by the controller with highest priority. They form the class of highest priority in the specification of the controller. Consequently, the priority of the class decreases with increasing its priority index. The non-deterministic choice between the failure actions cannot guarantee this. The effect is obtained by prioritising the failure actions:

$$Fail1 \parallel Fail2 \parallel Fail3 \leq Fail1 \ // \ Fail2 \ // \ Fail3$$

The generalisation of the made observations from the perspective of the program refinement is given by the following theorem:

Theorem 5. *The action system*

$$\mathcal{A} \mathrel{\widehat{=}} \textbf{const } c; \textbf{ global } z; \ [\![\ \textbf{var } pv, cv; \ I; \ \textbf{do } P \ ; (\mathit{Failure} \ // \ C) \ \textbf{od} \ ]\!]$$

such that

$$\mathit{Failure} \mathrel{\widehat{=}} \mathit{fail} \rightarrow \mathit{Emergency}$$

where fail is a predicate over the local and the global system variables and Emergency is equivalent to abort is refined by the action system

$$\mathcal{A}' \mathrel{\widehat{=}} \textbf{const } c; \textbf{ global } z; \ [\![\ \textbf{var } pv, cv; \ I; \ \textbf{do } P \ ; (\mathit{Failure}' \ // \ C) \ \textbf{od} \ ]\!]$$

where

$$\begin{aligned} \mathit{Failure}' \mathrel{\widehat{=}} \ & \mathit{fail}_1 \rightarrow \mathit{Emergency} \\ & // \ \mathit{fail}_2 \rightarrow \mathit{Rescue} \\ & // \ \mathit{fail}_3 \rightarrow \mathit{Degraded} \end{aligned}$$

and

$$\begin{aligned} \mathit{fail}_1 &\mathrel{\widehat{=}} \bigvee_{i=1}^{c_1} H_i \\ \mathit{fail}_2 &\mathrel{\widehat{=}} \bigvee_{i=c_1+1}^{c_2} H_i \\ \mathit{fail}_3 &\mathrel{\widehat{=}} \bigvee_{i=c_2+1}^{N} H_i \end{aligned}$$

and where H_i for $i = 1..N$ are the predicates over the local and the global variables such that fail $= \bigvee_{i=1}^{N} H_i$

Proof. The theorem follows from the observation that an action guarded by the disjunction of predicates can be partitioned to actions guarded by separate disjuncts. Moreover, the application of Rule 2 in Fig. 1 allows us to prioritise these actions. Finally, *abort* statement is trivially refined by any statement (by itself also as follows from the reflexivity of the refinement).

7 Concluding Remarks

We have shown how information about hazardous situations occurring in a plant can be embedded in the formal specification of a control program. Via this embedding the hazardous situations are treated according to their criticality and urgency. This allows to enhance safety of the overall system by ensuring that in case some marginal failure occurred simultaneously with a more critical failure the latter one will be treated with the highest priority. The development of the heater controller in Section 5 illustrated the application of the approach.

We have chosen to model the plant with the controlling software within the action system formalism. Our approach to embed safety analysis within the system development was based on using the refinement calculus associated with action systems. The creation of the system specification was carried out in the

stepwise manner: each refinement step incorporated information supplied by the corresponding level of the fault tree. Our example on the heater controller in Section 5 confirmed that the stepwise program refinement can naturally proceed hand-in-hand with the safety analysis. Observe also, the benefits of such an incorporation: the final form of the action modelling failures correctly prioritises the failures according to their criticality by the construction. A more elaborate case study on the approach is given in an accompanying paper [15] where we design a mine pump control system.

Further refinement steps are concentrated on the introduction of detailed specification of each identified hazard as illustrated in Section 5. Observe that applying the results of Theorem 5 we obtain a possibility to reason about each hazard in context of its own class. The reasoning structured in this way ensures a correct prioritising of failures causing hazards of different criticality. Therefore, when applying the techniques from Section 4 to elaborate on each of the identified hazards we do not only preserve the structure of the corresponding fault trees, but also the criticality of faults constituting the hazards from different classes.

Even though we in this paper concentrated on safety analysis and faulty behaviour of a system, the system itself is developed in a modular fashion, concentrating first on the normal behaviour of the system stating both the plant and the controller requirements within a single framework. Thereafter the different failure mechanisms are incorporated into the specification. Hence, we can separate the concerns, concentrate on parts of the system separately as well as use and state assumptions about the physical plant itself. This is an approach traditionally advocated by action systems [3, 10]. We as well as other researchers [7] argue that only such an approach makes a formal analysis of a system feasible, easily adjustable and less redundant.

Acknowledgements. The work reported here was supported by the Academy of Finland. The authors are grateful to the anonymous referees for their comments on the paper.

References

[1] R. J. R. Back and K. Sere. From modular systems to action systems. Proc. of *Formal Methods Europe'94*, Spain, October 1994. *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[2] R. J. R.Back and J. von Wright. Trace Refinement of Action Systems. In Proc. of *CONCUR-94*, Sweden, August 1994. *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[3] M. Butler, E. Sekerinski, and K. Sere. An Action System Approach to the Steam Boiler Problem. In Jean-Raymond Abrial, Egon Borger and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science Vol. 1165. Springer-Verlag, 1996.

[4] E.W. Dijkstra. A Discipline of Programming. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[5] K.M. Hansen, A. P. Ravn and V. Stavridou. From Safety Analysis to Software Requirements. In *IEEE Transactions on Software Engineering*, Vol.24, No.7, July 1998
[6] N.G. Leveson. Safeware: System Safety and Computers, Addison-Wesley, 1995.
[7] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. In *IEEE Transactions on Software Engineering*, 1994.
[8] Z. Liu and M. Joseph. Transformations of programs for fault-tolerance. In *Formal Aspects of Computing*, Vol 4, No. 5 1992, pp. 442-469
[9] A. McIver, C.C. Morgan and E. Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. In *Proc. of IRW/FMP'98*, Australia, 1998.
[10] E. Sekerinski and K. Sere (Eds.). Program Development by Refinement - Case Studies Using the B Method. Springer Verlag 1998.
[11] E. Sekerinski and K. Sere. A Theory of Prioritizing Composition . The Computer Journal, VOL. 39, No 8, pp. 701-712. The British Computer Society. Oxford University Press.
[12] K. Sere and E. Troubitsyna. Hazard Analysis in Formal Specification. In *Proc. of SAFECOMP'99*, France, 1999. To appear.
[13] N. Storey. Safety-critical computer systems. Addison-Wesley, 1996.
[14] E. Troubitsyna. Refining for Safety. TUCS Technical Report No.237, February 1999.
[15] E. Troubitsyna. Specifying Safety-Related Hazards Formally. In *Proc. of ISSC'99*, USA, 1999. To appear.
[16] K. Wong and J. Joyce. Refinement of Safety-Related Hazards into Verifiable Code Assertions. in *Proceedings of SAFECOMP'98,*, Heidelberg, Germany, October, 1998.