From System Safety Analysis to Software Specification

Elena Troubitsyna
Department of Computer Science
Åbo Akademi University
Lemminkaisenkatu 14 A
20520, Turku, Finland
+358 2 215 4086
Elena.Troubitsyna@abo.fi

ABSTRACT
In this paper we demonstrate how to derive software requirements from system safety analysis and capture them in a formal specification. We propose an integral approach for incorporating results of Fault Tree Analysis (FTA) and Failure Mode and Effect Analysis (FMEA) into the formal specification in the B Method. In our approach statecharts facilitate construction of the control system and serve as a basis for structuring and integrating results of FTA and FMEA. The use of statecharts as a communication media between safety and software engineers assists the process of requirements discovery. The use of formal development technique ensures correctness of implementing the requirements. The approach is illustrated by excerpts from the development of realistic industrial system – the liquid handling workstation Fillwell™.

Keywords
Safety, FTA, FMEA, statecharts, formal specification, B

1 INTRODUCTION

In this paper we aim at creating an integral method for formal software development which would naturally incorporate requirements from safety analysis [10,17] into the process of software construction. Currently the design environment of dependable systems is fragmented [9]. Indeed, even though safety analysis supplies critical requirements to be imposed on controlling software, safety analysis is yet not well integrated with software development process [11]. On the one hand, there is a traditional culture of isolating software development from hardware development and safety analysis. On the other hand, there is a lack of methods inter-relating dependability and software modeling.

The approach presented in this paper is a reflection on our experience gained while developing the liquid handling workstation Fillwell™ [3]. The development was undertaken as a part of the EU project MATISSE [12]. Within the project we carried out a formal development of controlling software for the workstation in the B Method [1]. We experienced a semantic gap between the results of safety analysis and software requirements. In this paper we propose to bridge the gap via statecharts modeling. The visual nature of statecharts provides a fruitful environment for the interdisciplinary communication and allows for a smooth transition from safety analysis to software requirements and then to the formal specifications in B.

In this paper we describe our approach to integrating results of FTA and FMEA [10,17] into the formal system specification. We demonstrate how not only safety invariant but also the mechanisms for error detection and recovery [2] can be derived from safety analysis and specified formally. We illustrate our approach by the corresponding excerpts from the development of Fillwell.

2 DISCOVERING SOFTWARE REQUIREMENTS WITH FTA

Safety analysis is devoted to identifying hazards, determining their causes, and deciding on their elimination and mitigation [10,17]. In this paper we focus on integration of two techniques – Fault Tree Analysis (FTA) and Failure Mode and Effect Analysis (FMEA) into the formal process of software development.

Fault Tree Analysis

FTA is a deductive safety analysis technique [10,17]. It is a top-down approach applied during all design stages. Preliminary hazard identification provides information about hazardous situations in functioning of the system. This information is taken as an input for the FTA. The result of the FTA is an identification of combinations of components states that result in hazards. Each fault tree has a root representing a hazardous situation. The tree traces the system to the lower component level to reveal the possible causes of failures.

To illustrate FTA let us consider an example – a simplified version of Fillwell [3] – a liquid handling workstation which was developed as a case study within the project MATISSE [12]. The workstation belongs to the class of products for drug discovery and bioresearch. The system consists of an operating head dispensing liquid substances into and aspirating them from micro-plates placed on a processing table. A gantry moves the operating head with high precision and speed from one plate to another in XYZ-directions. The head provides precise dispensing into high-

density micro-plates. The schematic representation of the workstation is given in Fig.1.

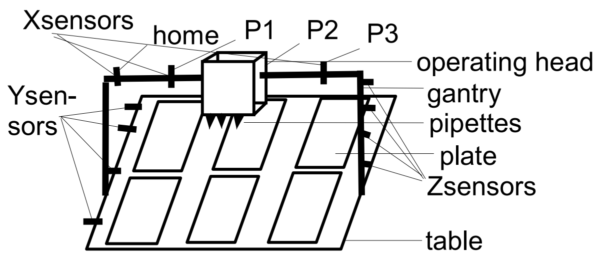

Figure 1. Schematic representation of Fillwell

The system's components *XCOMP, YCOMP*, and *ZCOMP* are responsible for moving the gantry and the operating head in X,Y,Z-directions correspondingly. The component *PUMP* aspirates and dispenses liquid by means of a peristaltic pump. The workstation is used to perform various pharmaceutical and bioresearch experiments. To perform an experiment a user defines a sequence of high-level commands – a protocol. Essentially a protocol consists of aspirating and dispensing liquid from one plate to another. The high-level commands are decomposed into sequences of lower-level commands activating system components in a certain order or in parallel.

The major safety and reliability requirement imposed on the system is to perform experiments correctly. Hence any situation which can potentially lead to an incorrect execution of the experiments is considered to be hazardous. For instance, among such situations is "damage of the equipment in the course of the experiment". It includes the states of the system in which safety operating boundaries are breached, the operating head collides with the table or the plates etc. In Fig.2 we present an excerpt from a high-level fault tree which traces the causes of the hazardous situation "head collides with plate" to the states of the system's components.

Let us note that the fault tree provides a logical representation of the hazardous system state in terms of the component states:

(<XCOMP active> \/ <YCOMP active>) /\ <ZCOMP active> (1)

Our experience shows that the fault trees for the component's states contributing to a hazard can be constructed on the basis of the pattern represented in Fig. 3.

While analyzing this pattern we observe that

- An invalid (or incomplete) safety invariant can result from a lack of integration of safety analysis into software development or from errors in translating results of safety analysis.
- Undetected software error breaching safety might result from a fault in verification process, e.g., a mistake in extracting a verifiable model from a program or an error in proofs can cause such an error.
- Absence of means to detect an error is a result of incomplete requirements which might again be a result of poor integration of safety analysis.
- Incomplete or incorrect means for fault tolerance are either results of missing safety requirements or errors in verifying fault tolerance mechanisms
- Hardware failure might be caused either by a multiple faults resulting in undetectable errors or by the primary hardware failures. It might result in inability of hardware to perform actions commanded by software.

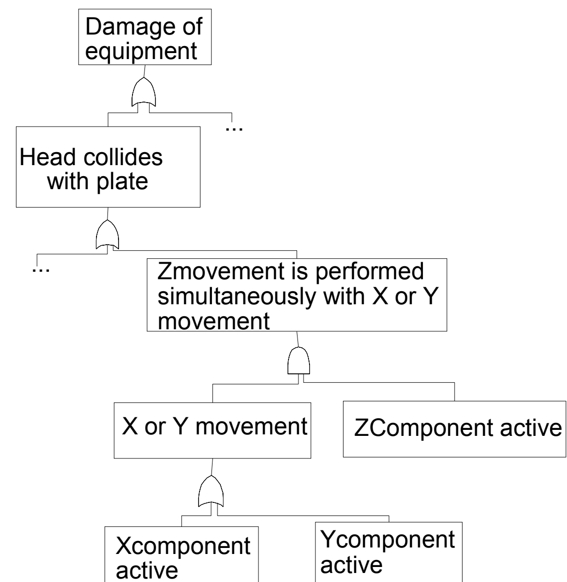

Figure 2. An example of a fault tree

Therefore we conclude that most of the software-related problems can be tackled by a method for a precise and unambiguous translation of results of safety analysis into formal software specification.

Often this problem is handled by giving fault trees formal semantics and then formulating safety invariant, which controlling software should preserve (e.g., [5]). However, this approach has a number of drawbacks. Firstly, not all the leaves of fault trees are related to software requirements (e.g., hardware failures should be addressed by introducing hardware redundancy), so an application of the approach is not always straightforward. Secondly, such an approach does not support inter-disciplinary communication because formal notation can be rather cumbersome to perceive for safety engineers. Finally, with such an approach it is rather difficult to correlate changes in software specification or

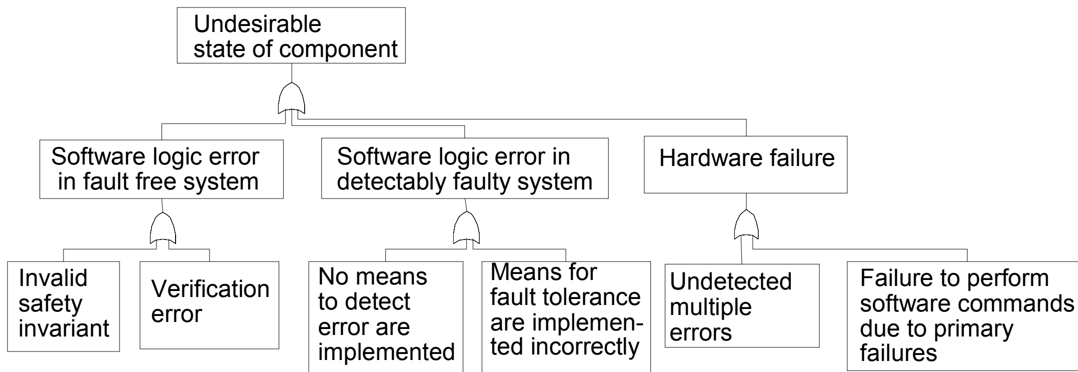

Figure 3. Fault tree of undesirable component state

system structure with safety. In this paper we propose an approach to tackling these problems.

From fault trees to software requirements via statecharts

It is widely accepted that visual techniques facilitate development of complex systems as well as interdisciplinary communication [4]. Our approach is based on using statecharts as means for integrating safety and functional requirements and translating them into a formal software specification. Statecharts [6] is a visual formalism which can be seen as a generalization of the conventional finite state automata. Statecharts form a part of a popular object-oriented modeling technique – Unified Modeling Language (UML) and are widely used nowadays. The formalism supports such features as hierarchy, concurrency and broadcast communication between system components.

Statecharts describe system's behavior as an evolution from one state into another upon arrival of an event. To illustrate statecharts modeling let us return to our example – the Fillwell workstation. The statecharts representation of the correct behavior of the system is given in Fig.4.

The system's components *XCOMP, YCOMP*, and *ZCOMP* and *PUMP* are represented as parallel activities. The parallel activities are rendered by splitting the system state with a dashed line. As a result each sub-state represents the behavior of the corresponding component. In general, the behavior of all the components follows the same pattern: while a component is idle (not engaged in executing any command) it can be activated. As a response to its activation a component executes a standard control loop (i.e., reading sensors and assigning actuators) to perform a requested command.

For example, let us analyze the behavior of *XCOMP* when it receives a request to move the operating head from its current position to the position *PXP1*. Assume that currently the operating head is in its *XHome* position. Placing the request is modeled by the event *XMove*. The request activates the component *XCOMP*. The active (engaged) state of the component is modeled by the state *XEngHome*. Upon leaving the position *XHome* the operating head switches off the home sensor *XHS* which is modeled by the event *XHSOFF*. The arrival of the operating head at the position *XP1* switches on the sensor *XSP1* which is depicted by the event *XSP1ON*. Since the requested destination is *XP1,* the condition [*xdest=XP1*] is satisfied and the component *XCOMP* arrives at the state *XIdleP1*. In this state *XCOMP* becomes inactive again. The behavior of the other components is modeled in the similar way. Due to lack of space we omit their detailed discussion.

Next we demonstrate how statecharts can assist in deriving safety invariant from the FTA. Let us note that the behavior of the entire system is represented by the statecharts depicting system's components functioning in parallel. At each instance of time each component is in a certain state. Hence, at each instance of time we can receive a "snapshot" of all components states. Let us also observe that lower level leaves of fault tree correspond to certain states of components. Hence, by matching these leaves with the corresponding component states we connect the fault tree with the statecharts model of the system. Furthermore, minimal cut sets of a fault tree logically describe the unsafe combinations of component states which lead to the hazard.

We illustrate this approach by expressing the hazard "damage of the equipment in the course of the experiment" in terms of component states. We extend the fault tree given in Fig.2. to incorporate the results of statecharts modeling. In Fig.5 we show an excerpt from the resultant fault tree. It represents the active state of *XCOMP* as a fault tree of its states. The hazard expressed on a high level (1) corresponds to the following logical expression over component states:

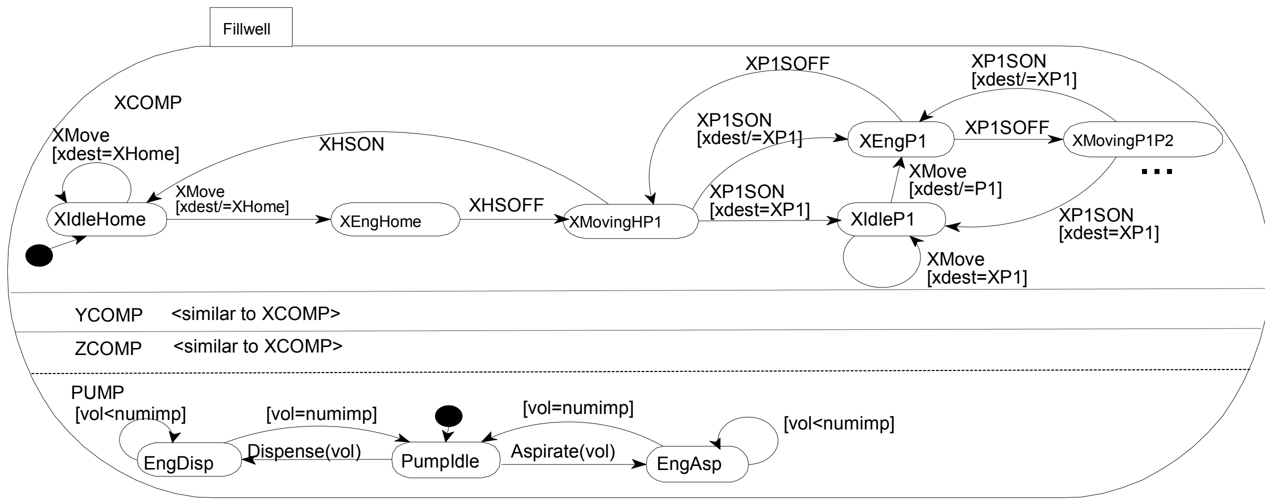

Fig.4. Excerpt from Fillwell statecharts

Hazard == (XCOMP=XEngHome ∨ XCOMP=XMovingHP1 ∨ XCOMP=XEngP1 ∨ XCOMP=XMovingP1P2 ∨ XCOMP=XEngP2 ∨ XCOMP=XMovingP2P3 ∨ XCOMP=XEngP3 ∨ YCOMP=…) ∧ (ZCOMP=ZEngHome ∨ ZCOMP=ZMovingHP1 ∨ ZCOMP=ZEngP1 ∨ ZCOMP=ZMovingP1P2 ∨ ZCOMP=ZEngP2 ∨ ZCOMP=ZMovingP2P3 ∨ ZCOMP=ZEngP3)

By negating this expression we obtain the safety condition which the controlling software should preserve to avoid the damage of the equipment:

Safety_cond= ¬Hazard

In the similar way the other hazards (e.g., "activating the pump while the operating head is moving") are traced to the components states. The overall safety invariant is formed as a conjunction over the logical expressions defining safety conditions to avoid the corresponding hazards.

As a result of integrating statecharts modeling and fault tree analysis we obtained a simple and unambiguous way to translate safety requirements into safety invariant. However, an application of the FTA alone is yet insufficient for completeness of safety requirements. Note that while constructing a fault tree we used the fact that, e.g., a component has failed and is in a certain state but we did not identify the means for error detection. Below we propose an approach to structuring and formalizing the results of FMEA which facilitate extracting the requirements for error detection and recovery.

3 EXTRACTING FAULT TOLERANCE REQUIREMENTS FROM FMEA

FMEA [10,17] is an inductive analysis method, which allows us to systematically study the causes of components faults, their effects and means to cope with these faults. FMEA is used to assess the effects of each failure mode of a component on various functions of the system as well as to identify the failure modes significantly affecting dependability of the system. FMEA supplies the information about failure modes of the individual components into the FTA. FTA and FMEA are often conducted together to compliment each other.

Even though FMEA should provide us with important requirements describing fault tolerance mechanisms often these requirements are rather difficult to extract. A major reason of this is a hardware-oriented style of FMEA resulting in ignoring the software aspect. Indeed, while safety engineers usually give a precise and detailed description of hardware-based fault tolerance mechanisms, they often describe software-based fault tolerance by a phrase like "modify software to detect error and implement error recovery" [17]. Obviously, such a description is too vague to result in precise software requirements.

In this paper we propose to tackle this problem by "enforcing" a structured way to describe software-based fault tolerance mechanisms on the basis of statecharts. In Fig.6 we present our proposal for integrating such a structured description into a traditional FMEA representation. Further description of the approach can be found in [18]. The FMEA table is extended with the explicit description of error detection mechanisms and structured description of the software implementation of remedial actions. To exemplify the proposed approach in

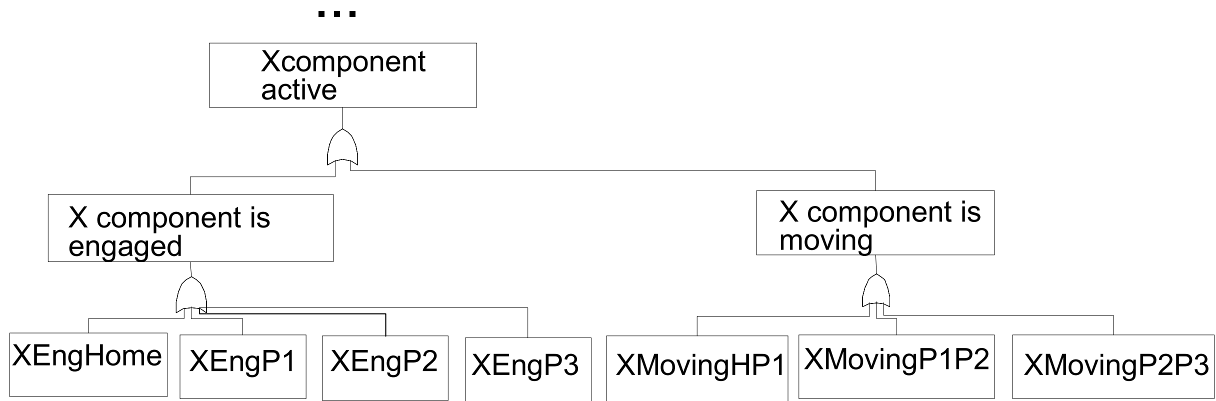

Figure 5. Lowest level of fault tree

Fig. 7 we present an excerpt from an analysis of the failure mode "stuck at zero" of the *XP1* sensor. The proposed extension enables a straightforward process of deriving requirements for software-based fault tolerance. Next we demonstrate how the requirements obtained from FTA and FMEA can be integrated with functional requirements in a formal specification.

4 INTEGRATING SAFETY ANALYSIS AND FORMAL SPECIFICATION

Correctness of controlling software is critical for system dependability [9]. Traditionally software correctness is ensured by an application of formal development methods. While selecting a formal specification framework we considered such criteria as simplicity of notation, availability of an automatic tool supporting development and verification and finally, possibility to automate translation from statecharts to the formal specification. As a result the B Method has been chosen.

Specifying in B

The B Method (hereinafter referred to as B) is an approach for the industrial development of highly dependable software [1]. The method has been successfully used in the development of several complex real-life applications [12]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [16], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping.

The development methodology adopted by B is based on stepwise refinement. While developing a system by refinement we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called *refinements*.

A formal specification is a mathematical definition of requirements imposed on a system. In B a specification is represented by a set of modules, called Abstract Machines. An abstract machine encapsulates a state and operations of the specification. Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialized in the INITIALIZATION clause. The variables in B are strongly typed by constraining predicates of INVARIANT clause. The constraining predicates are conjoint by conjunction (denoted as &). All types in B are represented by non-empty sets and hence set membership (denoted as :) expresses typing constraint for a variable, e.g., x : TYPE. Local types can be introduced by enumerating the elements of the type, e.g., TYPE = {element1, element2, …}.

The operations of the machine are defined in OPERATIONS clause. The operations are atomic meaning that once an operation is chosen its execution will run until completion without interference.

In this paper we model control systems as event-based systems. An event-based system consists of a set of the guarded operation SELECT cond THEN body END. Here cond is a state predicate, and body is a B statement describing how state variables are affected by the operation. When cond is satisfied the guarded operation is enabled, i.e., its behavior corresponds to the execution of the body. Any enabled operation – the operation with its condition being true – can be chosen for execution.

B statements that we are using to describe a state change in operations have the following syntax:

```
S == x := e | IF cond THEN S1 ELSE S2 END | S1 ; S2|
      x :: T | CHOICE S1 OR S2 OR…END | S1 || S2    |
```

The first three constructs - an assignment, a conditional statement and a sequential composition (used only in refinements) have the standard meaning. The remaining constructs allow us to model nondeterministic or parallel behavior in a specification. Nondeterministic assignment x :: T assigns variable x arbitrary value from given set (type) T. The statement CHOICE non-deterministically selects one of its branches for the execution. Finally, S1 || S2 models parallel (simultaneous) execution of S1 and S2.

Component	C: *Name of the component to which the unit belongs*			
Failure mode	*FM Description of the failure mode*			
Possible cause	*What causes the failure*			
Local effects	*How the failure effects the behavior of the unit or the component*			
System effect	*How the failure affects behavior of the overall system*			
Detection	*Aberrant event D_event arrives at D_state* *1) Detection state D_state: a state in which the arrival of detection event means that the failure occurred* *2) Detection event D_event: an erroneous signal received in a detection state* *Detection type: Timeout* *Timer Timer* *Time stamp TimeStamp* *Timing constraint Constraint* *Timer activated by A_event* *Timer deactivated by DA_event*			
Remedial action	*Compo-nent*	*Action*	*Software procedure*	*Effect*
	Name of compo-nent	*Description of an action to be performed*	*Yes / No*	*Description of the effect of action*

Figure 6. Extended FMEA

Component	XP1 sensor			
Failure mode	Stuck at zero			
Possible cause	Primary hardware failure			
Local effects	Sensor constantly sends zero signal			
System effect	Arrival of head at the position *XP1* is undetected			
Detection	Aberrant event *XSP2ON* (arriving at the position *XP2*) arrives at *XMovingHomeP1*(while the head is supposed to move to *XP1*) 1) Detection state *XMovingHomeP1* 2) Detection event *XSP2ON*			
Remedial action	Comp	Action	SW	Effect
	XCOMP	Stop. Return to *XP1*. If successful then retry current destination else stop system	Yes	Reference to statecharts describing the remedial action
	YCOMP	Suspend. If error recovery succeeds then continue	Yes	Pause YCOMP. If error recovery succeeds then continue from current position

Figure 7. Example of extended FMEA

The special case of a parallel composition is a multiple assignment which is denoted as x,y := e1,e2.

Modelling control system in B

The process of translating statecharts representation of system's behaviour into B is rather straightforward. In the abstract machine we represent concurrent components, superstates, states and events of statecharts as corresponding state variables. Initial states in statecharts correspond to initialization of the appropriate variables of the B machine. Transitions between states are modeled by the operations of abstract machine. In general, the transition from the state S in the superstate SS to the state S' in the superstate SS*'* upon arrival of the event E can be specified by an operation of the following form:

```
OP = SELECT EVENT=E & SUPERSTATE=SS &
                STATE=S
        THEN SUPERSTATE:=SS'  || STATE:=S' END
```

Conditions are expressed as predicates over state variables. An operation of the form

```
OP = SELECT Event=E & Superstate=SS & state=S
        THEN  IF  cond1
                    THEN Superstate:=SS'  || state:=S'
                     ELSEIF cond2
                     THEN Superstate:=SS'' ||  state:=S''
                      END
                      END
         END
```

models a transition from the state S in the superstate SS to the state S' in the superstate SS*'* upon arrival of the event E if the condition cond1 holds and to the state S" in the superstate SS" upon arrival of the event E if the condition cond2 holds. An elaborated description of translating statecharts into B can be found in [13].

Usually a control system is a reactive system with two main entities: a plant and a controller. The overall behavior of the system is an alternation between the events modeling plant evolution and controller reaction. As a result of the initialization, the plant's operation becomes enabled. Once completed, the plant enables the controller. The controller monitors the behavior of the plant and adjusts it to provide intended functionality and maintain safety. The controller is specified as a composition of operations modeling

- routine control, i.e., reaction of the controller on the normal (fault-free) and safe behaviour of the plant

- error detection, i.e., the operations which are enabled when erroneous state of the system is detected

- remedial actions, i.e., the operations which specify the behaviour of the controller while recovering the system from errors

The general structure of a control system which we propose is given in machine ControlSystem

```
MACHINE
  ControlSystem
VARIABLES
  flag, state_variables
INVARIANT
  flag : {pl,contr}  & safe
INITIALIZATION
  flag := pl …
 …
OPERATIONS
  Plant = SELECT flag=pl THEN generate_stimuli ||
                        flag:=cont END;
  Control = SELECT flag= contr & safe & plant_stimulus
               THEN control_action || flag :=pl END;
  Detection = SELECT flag=cont & error_detected
                  THEN  Initiate remedial actions END ;
  Remedy= SELECT flag= contr & plant:stimulus
               THEN remedial_action || flag :=pl  END
END
```

The safety invariant safe is obtained by merging via conjunction safety conditions obtained as a result of FTA. The plant is specified as a non-deterministic choice of events. Such a specification allows us to model not only normal behaviour of the plant but also errors which manifest themselves as aberrant events or timeouts. The routine control is specified by the set of operations of the general form Control. The operation Control is enabled if the plant is safe and has generated stimulus corresponding to the fault-free behaviour. In contrast the operation Detection becomes enabled if one of the error detection mechanisms has been trigged by the stimulus obtained from the plant. The predicates error_detected in the guards of Detection operations are obtained from the extended FMEA description of the error detection mechanisms. In our approach the predicates defining detection conditions given as aberrant events have identical form:

Event=D_event & state=D_state

obtained by a straightforward translation of the information given in "Detection" row of extended FMEA. We model error detection by timeout in the similar way:

Timer=ON & t-TimeStamp>Constraint

where t is a variable modeling time.

However, this requires activation/deactivation of timer by the normal control operations which are enabled when the events A_event and DA_event arrive. The activation of timer Timer := ON and deactivation of timer Timer := OFF are added to these actions as simultaneously executed statements.

Finally, the remedial actions are of the form Remedy. They are activated upon detection of corresponding errors. The operations specify the reaction of the controller on the stimuli of faulty plant in the process of error recovery. Upon successful completion of error recovery the controller resumes normal control, i.e., the operations Control become enabled again. Upon failure of error recovery the remedial actions shut down the system.

Fillwell: Formal B Specification

To illustrate formal modeling of discrete control systems in B, in Fig. 8 we present an excerpt from the specification of Fillwell. The system is specified according to the general specification form ControlSystem. The plant is modeled by the operation XPlant. The operations XC_i,i:1..N specify routine control operations to be provided by the component *XCOMP* in response to stimuli of fault-free plant. We assume that the reaction of the controller takes negligible amount of time so the controller can react properly on changes of the plant state.

The operation XP1SF_D specifies detection of the error *"XSP1 stuck at zero"*. Observe that the specification is obtained by the straightforward translation of the description of the detection mechanism. The corresponding remedial actions XP1SF_R1.. XP1SF_R4 are obtained on

```
MACHINE
  XFillwell
SETS
  XIEVENTS : { XHSOFF,XP1SON,…};
  XEEVENTS: {XMove};
  XSTATE : {XIdleHome, XEngHome,XMovingHP1, XIdleP1,… };
  XDEST : {XHome, XP1,XP2,XP3};
VARIABLES
  fl, xe, xst, xdest, fdest…
DEFINITIONS
  safe = safety_cond1 & safety_cond2 &
INVARIANT
  fl : {cont,pl} & xe : XIEVENTS \/ XEEVENTS &
  xst : XSTATES & xdest, fxdest : XDEST & safe
OPERATIONS
  XPlant = SELECT fl= pl  THEN
              CHOICE xe : XIEVENTS
              OR xe := XMove || xdest : XDEST END  || fl:= cont
  XC_1   = SELECT fl=cont & xe= XMove & xst= XIdleH & safe
              THEN IF xdest = XHome THEN skip
                   ELSE xst := XEngHome || fdest := xdest END
             || fl := pl END
  XC_2 = SELECT fl=cont & xe= XHSOFF & xst= XEngHome &
                   safe
           THEN xst := XMovingHP1 || fl := pl END
  XC_3 = SELECT fl=cont & xe=XP1SON & xst=MovingHP1 &
                   safe
           THEN IF xdest  = XP1 THEN xst:= XIdleP1
                ELSE xst := XEngP1 END || fl := pl END
  XP1SF_D = SELECT fl=cont & xe=XP2SON & xst=XMovingHP1
              THEN xst := XP1SF_DET || xe,xdest:= XMove,XP1||
                 PauseY END
  XP1SF_R1 = SELECT fl=cont & xe=XMove & xst= XP1SF_DET
               THEN xst := XP1SF_Eng || fl := pl
  …
  XP1SF_R3= SELECT fl=cont & xe=XP1SON &
               xst= XP1SF_MovingP2P1
             THEN xst := XIdleP1 || xe,xdest  := XMove,fdest END
  XP1SF_R4= SELECT fl=cont & xe=XHSON & xst=
                 XP1SF_MovingP2P1 THEN xst :=XStopped END
END
```

Fig.8. Excerpt from formal specification of Fillwell

the basis of statecharts representation of the recovery action (we omitted the presentation of these statecharts – they merely define course of events in the recovery action). The recovery action XP1SF_R4 is executed in case of failure of error recovery – it brings the component in a non-operational state XStopped.

The initial formal specification contains nondeterminism and abstract data types. They are replaced by implementable constructs in the process of system refinement. The final refinement step decomposes the system into a plant and controller, i.e., allows us to arrive at the specification of the controlling software as such. Finally, executable code is generated.

The process of obtaining B specifications from statecharts is facilitated by the U2B tool [15]. Moreover, we are developing a tool which supports FTA and extended FMEA and integrates their results into the B specifications as described above.

5 CONCLUSIONS

In this paper we presented an approach to integrating safety analysis with formal development. The approach enables a smooth transition from reasoning about safety to specifying controlling software. Our specifications are formal and can be (automatically) transformed into executable programs through a succession of refinement steps. In this paper we omitted a detailed description of refinement process (it can be found, e.g., in [7,8,14]). Instead we focused on capturing requirements supplied by FTA and FMEA in the formal specifications. The use of statecharts allowed us to inter-relate software and safety modeling and significantly simplified derivation of safety-related requirements. The various stages of the approach have been automated and allowed us to scale up formal development.

In our future work it would be interesting to integrate the other safety techniques with formal modeling and eventually defragment the design environment of dependable systems.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. T.Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice.* Dependable Computing and Fault Tolerant Systems, Vol 3**.** Springer Verlag; 1990.
3. *FillwellTM 2002- Feature guide.* Via http:/lifesciences.perkinelmer.com/.
4. A.Hall. Formal methods start to add up once again. In *Computing,* 8th January 2004.
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. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231-274, 1987.
7. L. Laibinis, E. Troubitsyna. Refining fault tolerant control systems in B. In Proc. of SAFECOMP'2004. Potsdam, Germany, LNCS, Springer-Verlag. To appear.
8. L. Laibinis, E. Troubitsyna. Fault Tolerance in a Layered Architecture: A General Specification Pattern in B. In Proc. of SEFM'2004. IEEE Computer Press. Beijing, China. To appear.
9. J.-C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Vienna, 1991.
10. N.G. Leveson. *Safeware: System Safety and Computers,* Addison-Wesley, 1995.
11. R. Lutz, H.Y. Shaw. Applying Adaptive Safety Analysis Techniques. In Proc. of *10th International*

Symposium on Software Reliability Engineering. Boca Raton, Fl, November 1999.
12. *MATISSE Handbook for Correct Systems Construction*. EU-project MATISSE: Methodologies and Technologies for Industrial Strength Systems Engineering, IST-1999-11345, 2003. *http://www.esil.univ-mrs.fr/~spc/matisse/Handbook/*.
13. E. Sekerinski. Graphical Design of Reactive Systems. In D.Bert (Ed): Proc. of *2nd International B Conference*. Lecture Notes in Computer Science 1393, pp. 182-197, Montpellier, France, 1998.
14. K. Sere and E. Troubitsyna. Safety Analysis in Formal Specification. In J. Wing, J.Woodcock, and J.Davies (Eds.), *FM'99. Proc. of World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1709, pp. 1564-1583, Toulouse, France, 1999.
15. C. Snook and M. Butler. U2B - UML to B translation tool and Manual V4.4 available at http://www.ecs.soton.ac.uk/~cfs/U2Bdownloads/U2Bdownloads.htm.
16. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 2001. Available at http://www.atelierb.societe.com/index uk.html.
17. N. Storey. *Safety-critical computer systems*. Addison-Wesley, 1996.
18. E. Troubitsyna. Integrating Safety Analysis into Formal Specification of Dependable Systems. In *Proc. of Annual IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems.* Nice, France, April 2003.