SAFETY OF COMPUTER CONTROL SYSTEMS: CHALLENGES AND RESULTS IN SOFTWARE DEVELOPMENT

Janusz Zalewski[*] **Wolfgang Ehrenberger**[**] **Francesca Saglietti**[***] **Andrew Kornecki**[****]

** University of Central Florida*
Orlando, FL 32816-2450, USA
jza@ece.engr.ucf.edu
******* *University of Applied Science*
36039 Fulda, Germany
wolfgang.ehrenberger@informatik.fh-fulda.de
**** Institute for Safety Technology, Forschungsgelände*
85748 Garching / Munich, Germany
SAF@grs.de
***** Embry-Riddle Aeronautical University*
Daytona Beach, FL 32114, USA
korn@db.erau.edu

Abstract: This paper reviews some results in improving software safety in computer control systems. The discussion covers various aspects of the software development process, as opposed to the product features. Software diversity, off-the-shelf software, rigorous and formal software development are discussed.

Keywords: Software safety, software diversity, off-the-shelf software, rigorous development, formal verification, UML, software for computer control.

1. INTRODUCTION

The key aspect of safety in computer control systems is to minimize the risk of harm, which can lead to loss of life, limbs, or large financial losses due the failure of hardware or software of a computer control system embedded in a larger application. Safety aspects are important in all modern computer applications, where computer control is embedded in larger systems, for example: all means of transportation, such as cars, railways, airplanes, ships; large plants, such as nuclear power plants, chemical plants, etc.; as well as smaller devices, such as medical electronic devices.

New challenges show up every day, because computer-controlled systems are being increasingly applied in other areas related to human safety. In addition to traditional applications concerning safety listed above, several new areas have emerged, related to safety of computer control, including: telecommunication systems, banking systems, fire protection systems on oil/gas platforms, and others. Assuring safety becomes more and more important, because of an increasing concern about safety of computers and computer-controlled systems in a modern society as a whole.

The technologies of providing safety assurance in computer control systems are often based on the architecture of feedback control, but they rarely use, if at all, results from continuous systems. Due to the discrete nature of the safety problem, methods of discrete mathematics are usually applied.

In this view, verification of hardware and software, with respect to safety, is a key challenge. Methods currently offered are so complex that they are manageable only for simple systems, while the complexity of safety-related applications is usually high and continues to increase dramatically with the progress of computing technologies.

In this paper, we present selected approaches to increase safety of computer control systems via the appropriate software design process. We focus on the development process to improve software safety, as opposed to engineering the design structure of the controller itself.

2. SOFTWARE SAFETY: PRODUCT VERSUS PROCESS

Safety is usually defined as a 'negative' property that asserts simply that nothing bad happens (Rushby 1994). Safety is addressed through a hazard analysis process, which is normally not conducted in non-critical software development. Attention of the developers must be focused on preventing hazards that are conditions that lead to a mishap, rather than preventing mishaps directly.

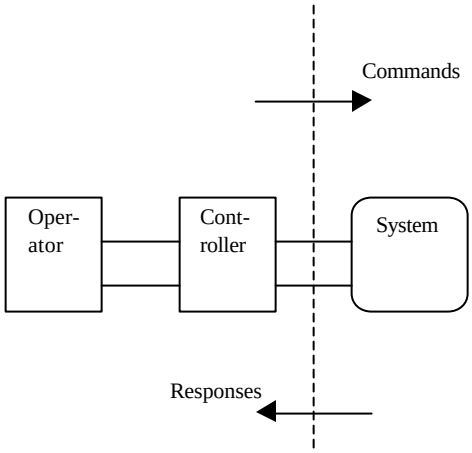

Fig. 1. Context for Discussing Safe Software.

In terms of a control system (Fig. 1), hazards are caused by various failures of the controller. An 'omission failure' is caused by a failure of the controller to react correctly to a given system state. The controller providing incorrect commands or inputs to a system causes a 'commission failure'. The controller responding correctly but outside of the required timing constraints causes a 'timing failure'.

There have been many technical solutions proposed to ensure software safety (Leveson 1996, Hilburn and Zalewski 1996). Some of the authors' own research (Anderson et al. 1999, Sahraoui et al. 2000) included the development of a software safety shell, that depends on a guard to ensure a safe behavior of the system (Fig. 2). The guard is a low-level construct that detects the danger and then forces the system to go to a safe state instead of hazardous one, if at all possible. This approach was applied in a traffic-light control case study to detect the conditions that, if unchecked, would lead to a failure.

All theoretical methods, which rely on engineering the proper design of the controller's structure to detect hazardous states, however, ultimately need to go through the software development process. This means that just designing the controller is necessary but not sufficient for ensuring safety, and during software development additional care has to be taken to retain or even improve safety properties embedded in the design.

One traditional method of developing software for safety is to use diversity, that is, have independent teams apply different principles to develop the software product to the same specifications. Even this simple approach, however, has its flaws, which we discuss in Section 3. A further problem arises when re-using software components, typically commercial off-the-shelf (COTS) software, which nowadays is more and more often applied in safety-related systems. New principles of dealing with COTS in safety-related systems have to be developed, which we discuss in Section 4.

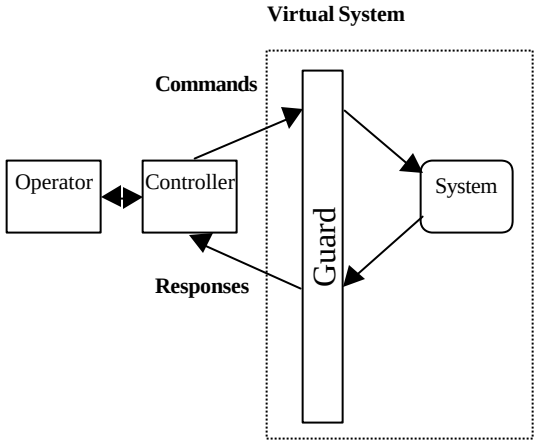

Fig. 2. System View of a Safety Guard.

Applying diversity during development and using systematic methods for dealing with COTS software might increase the confidence in software correctness, but in general are not sufficient to guarantee full product safety. In order to further improve software safety and lower the risk associated with its use, approaches relying on more formal mathematical techniques are required. One step in this direction is to support the design phase in the development process by formal design verification, which makes the entire process more rigorous.

We discuss it in Section 5 in relation to Petri nets. However, purely formal techniques are not sufficient to capture completely the reality to be modeled and analyzed; a very promising approach to the development of safety-related software consists of combining formal methods with established engineering techniques throughout the entire software life cycle. We discuss this issue in Section 6, for verifying timing requirements with UML and extended timed graphs.

The rest of this article presents an overview on the techniques mentioned above; in the following four chapters each of the co-authors summarizes the results and insight gained in his / her own area of research.

3. DIVERSITY IN THE SOFTWARE PROCESS

3.1 Common Failures in Diverse Variants

Since a long time it has been of interest whether independently developed variants of software could contribute to the increase of safety. This concept has been particularly attractive to reduce the licensing costs of safety-related programs. The basic idea has been as follows:

- Step 1. Produce independently two variants of software to solve the same task.
- Step 2. Demonstrate that each single variant has achieved a certain reliability, for example, that its failure probability per demand $p_1 \approx p_2$ is below a certain limit, say 10^{-4}.
- Step 3. Conclude for the probability of common failure per demand of both variants

$$p_{12} \approx p_1 * p_2 \approx p_1^2 < 10^{-8} \qquad (1)$$

This way of thinking may lead to commercially attractive solutions under certain circumstances. Doubling the effort - we have to produce two variants - would result in the *square of the relevant reliability figure*. In other words, *linear effort increase during production could provide exponential reliability gain*. The gain would improve safety properties, at the expense of a loss of availability properties. Under such assumptions diversity looks to be economically attractive.

Of course, during producing two variants one would take any precaution to avoid common faults among variants. The collection of diversity techniques proposed in (Saglietti et al. 1992) could be taken to select a proper set of techniques.

There were, however, severe objections against conclusion (1). In a theoretical work (Eckhardt and Lee 1985) it was shown that (1) is to be augmented by a measure of the varying "difficulty" throughout the input data space, namely the variance of the probability of committing programming errors over the input domain of the two diverse program variants:

$$p_{12} \approx p_1^2 + \mathrm{Var}(q) \qquad (2)$$

where q represents the probability of an arbitrary variant failing on input x. The variance is to be taken over all inputs x. An accurate estimate of q would require examination of a large number of software variants for any practical project, which is obviously not viable. Var(q) may exceed p_1^2 significantly.

Further investigation leads to the result derived in (Ehrenberger and Saglietti 1993). If the input domain of a diverse software system is processed by K disjoint input channels, and if the software of the individual channels fails independently, the probability of common failures is increased by a factor close to K:

$$p_{12K\mathrm{Channels}} \approx p_{12} * K \approx p_1^2 * K \qquad (3)$$

3.2 Estimation of the Variance

Littlewood and Miller (1987) developed equation (2) further and a more general result has been derived, including the particular case of *dissimilarity being enforced* during the development of the diverse variants:

$$p_{12} = p_1^2 + \mathrm{Cov}(q_1, q_2) \qquad (4)$$

where $q_1(x)$ and $q_2(x)$ represent average probabilities of failure on input x of two arbitrary variants developed by using different methodologies (forced diversity). Formula (4) degenerates to (2), if no dissimilarity was enforced. In principle, $\mathrm{Cov}(q_1,q_2)$ can become negative. Also here no estimate of the size of the covariance is known for an arbitrary concrete system.

Further details on theoretical considerations are presented in (Ehrenberger 2001b). As they do not provide a formal proof, experimental evidence is needed for support. As mentioned above, such evidence is difficult to get from software development, because programming a large number of variants would be too expensive. However, there may be a way out.

If we believe that *programming is a human activity similar to writing exams*, we may consider the results of student exams in place of the results of programs. Table 1 shows the envisaged analogies. The table of the exam results is given in (Ehrenberger 2001a), where 8 questions had to be answered, including some on deriving algorithms. A total of 13 exam papers were considered worth being used in the evaluation, because the number of faulty answers was reasonably small. They formed 78 pairs.

Table 1 Analogies between programming faults and faults in exams.

Aspect	Solution of exam	Program
Requirements from	question sheet from professor	program specification
Result	exam paper	developed code
Form of requirement	answer question x	react on input x
Faults	faults in solving the question	program faults
Form of failure	human failure	human failure
Single task	single question, single answer	single program part
Single fault	fault in answer x	fail. on input x
Characteristic: probability of fault per question or program part	probability of fault per question $\frac{\text{number of incorrect answers}}{\text{total number of questions}}$ each answer, which does not score fully, is considered faulty	probability of failure per program part: $\frac{\text{number of failing program parts}}{\text{total number of program parts}}$ each program part, which has a fault is faulty
Characteristic: probability of fault for an answer or an input portion	fault characteristic of an answer per exam: $\frac{\text{number of exam papers with faults in this answer}}{\text{total number of exam papers}}$	fault characteristic per input portion: $\frac{\text{number of variants with faults in this input area}}{\text{total number of programs}}$
Variance to be calculated	over all answers, over the probability of the faulty answer	over the individual input domain portions, over the probability of faulty treatment of an input

Any correct answer was qualified with 0, any incorrect one with 1. For each question the number of failed answers was counted. The q_i of question i was calculated as:

$$\frac{\text{number of exam papers with faults in answer i}}{\text{number of all exam papers}}$$

and the variance of the difficulty of the questions, with N representing the number of questions:

Var(question difficulty) =

$$\frac{1}{N}\sum_{i=1}^{N}(q_i - \frac{1}{N}\sum_{i=1}^{N} q_i)^2 \qquad (5)$$

On the other hand, the probabilities of common faults of two students (i,j) can be taken from the related matrix as p_{ij} equal to the following:

$$\frac{\text{number of answers with faults in both exam papers}}{\text{number of all questions}}$$

The probabilities of individual faults per solution p_i and p_j could also be evaluated:

$p_{mean} = 0.365$; $(p_{mean})^2 = 0.133$

Var(question difficulty) = $0.023 < (p_{mean})^2$.

$p_{ij} < p_i * p_j$ for 28 pairs

$p_{ij} = p_i * p_j$ for13 pairs

$p_{ij} > p_i * p_j$ for 37 pairs

$p_{ij} \leq p_i * p_j$ for 41 pairs

$p_{ij} \leq 2 * p_i * p_j$ for 70 pairs

These observations together with the theoretical considerations suggest the following estimate for the common failure probability of a two-fold diverse system:

$$p_{12} \approx p_1^2 + Var(q) \approx 2*p_1^2 \qquad (6)$$

Regarding the exams' results one has to remember that no type of forced diversity had been applied. As forced diversity will be the rule for any real project with diverse software, failure independence is likely to be higher than in the exams.

3.3 Results

If one accepts the *analogy of the fault producing processes in students' exams and programming,* one may consider data from university exams as a basis for studying software diversity. The investigation of the answers of a student exam shows that in general the intuitive result (1) is too optimistic. A correction by the factor 2 is appropriate. This correction is more conservative than result (4).

If the program reduces information during its execution by a selection mechanism, the number of channels involved in that reduction has to be considered as a further source of common failure of independent variants. In many cases of industrial software application the influence of the information reduction will be more substantial than the influence of the varying difficulty of the input domain. Under ideal circumstances and bringing together results (6) and (3), we may conclude that:

$$p_{12,KChannels} \approx p_{12} * K \approx 2 * p_1^2 * K \qquad (7)$$

Under ideal circumstances the probability of common failure of two diverse software variants may be estimated as two times the square of the value of the failure probability of the singular variant times the number of diversity channels that entail information reduction.
The results of the experiment on diverse software development can be summarized as follows:

- Errors are not made independently of the questions to be answered.
- The value of the variance of the difficulty over the input domain is close to the value of common failure probability.
- Under ideal circumstances the probability of common failure equals the square of the failure probability of one variant multiplied by the doubles number of channels.

4. OFF-THE-SHELF SOFTWARE

For evident economical reasons, COTS software components are increasingly (re)used in different application areas, including safety-critical ones. Due to their origin, the available information on the underlying development process of these components is often only fragmentary.

A very limited knowledge on the component production, as well as differences between past and future usage (such as, different reliability demands or different usage profiles) create a serious challenge for the software engineering community facing the problem of assessing the suitability of COTS components for new development projects.

The strategy suggested to approach this difficult problem consists of five successive decision phases, as illustrated in Figure 3:

- Phase 1. Identification of safety demands at system level;
- Phase 2. Analysis of role of COTS software within the system (safety relevance & sensitivity);
- Phase 3. Qualitative (subjective) assessment of the software process and product quality;
- Phase 4. Quantitative (objective) assessment of past (testing &) operating experience;
- Phase 5. Validation of component interfaces within the integrated system.

Considerations on phases 2, 3 and 4 were presented in (Saglietti 2001, Saglietti 2000b and Saglietti 2000a), respectively. Each phase is discussed below in terms of key factors to be considered and necessary actions to be taken during development.

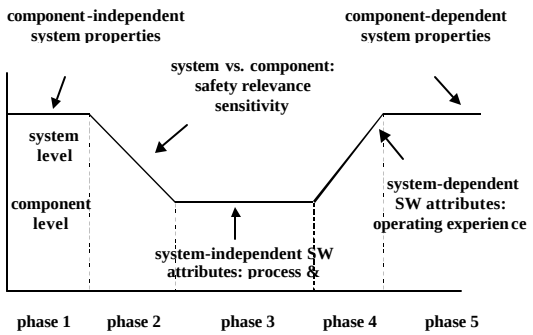

Fig. 3. Phased Approach to Component Analysis.

Phase 1: Identification of safety and reliability demands at the system level.

Key factors: risk analysis (black-box, component-independent).

- Risk analysis: identification of occurrence of failure-initiating events and loss caused by them; comparison of hazards involved in automation with those inherent to the application as such, i.e., in an uncontrolled mode; identification of critical events to be controlled; classification of event criticality if not under control.

- Analysis of results: determination of minimum quantitative target for reliability demands to be demonstrated, in terms of minimum probability of operational survival, i.e. of correct or acceptable performance under application-specific operational demand profile.

Phase 2: Analysis of the role of COTS components within the system (Saglietti 2001).

Key factors: COTS safety relevance, sensitivity (objective, quantitative, system dependent), safety criticality and sensitivity.

- Safety Criticality: identification of potentially critical failure propagation through COTS modules, evaluation of individual responsibility of COTS components in terms of safety-relevance to distribute verification and validation effort accordingly, determination of bottlenecks for system safety, definition of fault tolerant architectural properties accordingly.

- Sensitivity: determination of impact of component reliability on system reliability to know whether modest reliability figures at component level are acceptable for systems with higher reliability demands, in order to derive to which extent a post-qualification of pre-developed software allows to expect an increase of system reliability.

Phase 3: Qualitative assessment of COTS at component level (Saglietti, 2000b).

Key factors: COTS process and product (subjective, qualitative, system-independent), assessment of non-operational evidence.

Qualitative judgment on non-operational aspects from all life-cycle phases preceding operation, should include the following aspects:

- Development process (structured, semi-formal, formal methods)
- Safety culture (awareness of consequences, ethical and financial)
- Documentation (accuracy and consistency of reports and dependencies)
- Resources (human labor and mechanical aid)
- Informal checks (manual analysis of code and documents)
- Automated static analysis (syntactic and semantic checks by automatic tools)
- Non-operational tests (execution for non-representative inputs).

Phase 4: Quantitative assessment of COTS components within the system (Saglietti 2000a).

Key factors: COTS behavior during testing and operation (objective, quantitative, system-dependent), assessment of operational evidence.

- Quantitative judgment of operational aspects: in spite of deterministic nature of logical faults, a probabilistic approach is justified by input randomness, representing physical variables subject to unpredictability of state transitions in technical processes under control.

- Estimation of software reliability by statistical testing on the basis of significant amount of operational evidence, correct execution of a large number of independent, operationally representative scenarios; based on sampling theory, estimation of upper bound of failure probability at a given confidence level.

Phase 5: Validation at integrated system level.

Key factors: interface analysis (white box, component dependent, context dependent).

Inconsistencies between physics and logical specification have to be analyzed, including:

- Violated global properties, such as in Airbus incident in Warsaw.

- Violated local properties at interfaces, such as:

  (a) Mars Climate Observer, where force reference system has not been specified by NASA; JPL controllers expected Newton (British system), Lockheed Martin Astronautics provided metric units.
  (b) Berlin Fire Brigade – Millennium: different operating systems in different system parts, inconsistent communication of date format.
  (c) Ariane 5 Explosion, where conversion routine originally developed for flight trajectory of Ariane 4 used insufficient range.
  (d) Blood Databank, where program designer developed the database manager for a single computer application without considering networked applications; could not anticipate that simultaneous access to a record by two users would lead to a hazard.

5. RIGOROUS DEVELOPMENT PROCESS

The fundamental objective of achieving software safety is to guarantee that the software does not cause or contribute to a system reaching a hazardous state. This may be supported by the rigor and conformance to well defined engineering standards.

5.1 Safety Analysis and Verification

Mojdehrakhsh et al. (1994) describe a process for safety improvement. The process must assure that the software safety analysis is closely integrated with the system safety analysis and the software safety is explicitly verified.

The following activities for software safety program are geared toward achieving the stated objective, with emphasis on analysis first and verification next. The analysis phase involves conducting system safety analysis to:

- Identify the key inputs into the software requirements specification, such as hazardous commands, limits, timing constraints, sequence of events, voting logic, failure tolerance, etc.
- Create and identify the specific software safety requirements in the body of the conventional software specification.
- Identify which software design components are safety critical.

The initial analyses and subsequent system and software safety analyses identify when software is a potential cause of a hazard or will be used to support the control of a hazard. Using specific software design and implementation techniques and processes is crucial to reduce potential hazards introduced by software. The associated verification phase involves the following steps:

- Apply specific verification and validation techniques to ensure appropriate implementation of the software safety requirements.
- Create test plans and procedures to satisfy the intent of the software safety verification requirements.
- Introduce any necessary corrective actions resulting from the software safety verification.

The process is fundamental to the identification of the safety-critical functions and the causal factors, including the factors that may be software-induced or controlled. The identification shall include also creation of an appropriate taxonomy of hazards (commission, omission, timing error).

Some of the techniques used for analysis and verification include (Bishop, 1990): Fault Tree Analysis (FTA), Event Tree Analysis (ETA), Hazard Operability Analysis (HAZOP), Failure Mode Effect Analysis (FMEA), Failure Mode Effect and Criticality Analysis (FMECA), Common Mode Failure Analysis (CMF), Cause Consequence Diagrams (CCD), and Petri Nets (PN).

Most of these methods allow for rigorous treatment of the development process. The one we found particularly suitable for studying safety properties is Petri nets (Saglietti 1998, Kornecki at al. 1998). We have applied it to the analysis of the TCAS software specification.

5.2 Petri Nets in Design Verification

TCAS is a Traffic Collision Avoidance System, introduced to reduce the risk of mid-air collisions between aircraft. The TCAS equipment is a small electronic device, consisting of a computer and software, a directional antenna, a transponder and cockpit display and controls (Fig. 4 and 5).

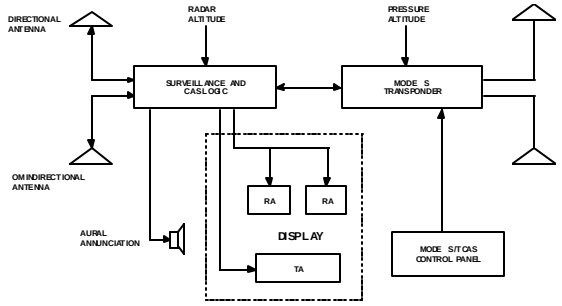

Fig. 4. TCAS Architecture.

TCAS continuously monitors aircraft within 10 nautical miles to identify the potential threat. If an aircraft is determined to be a threat, a Traffic Advisory is issued showing the available information about the potential threat, including the call sign, relative position, altitude difference, and altitude vector.

Fig. 5. TCAS Cockpit Display

If an aircraft persists to be a threat, the TCAS evaluates whether collision might occur within the next 23 seconds. In such case, a Resolution Advisory is issued, which requires the protected aircraft to execute one of the range vertical maneuvers.

Based on the TCAS specification, a Petri net model was built and verified using a public domain tool Cabernet (Kornecki et al. 1998). The objective was to check formally, whether certain states, having impact on safety, can be achieved as a result of inconsistencies in the specification.

For example, state changes to the Emergency State, described under respective conditions, as below, converted to predicates:

IntruderStatus = Continuing &
Threat = Establishes &
RA = VerticalSpeedLimit &
Intruder is TCAS-equipped &
Own Mode S ID is higher &
Threat has the sane sense as Own &
!(RA display deferred)

were analyzed using the reachability graphs. In effect, several properties were verified with respect to safety aspects. Verification of a sample property of the original TCAS specification, that has been found “not-satisfied”, is shown in Fig. 6.

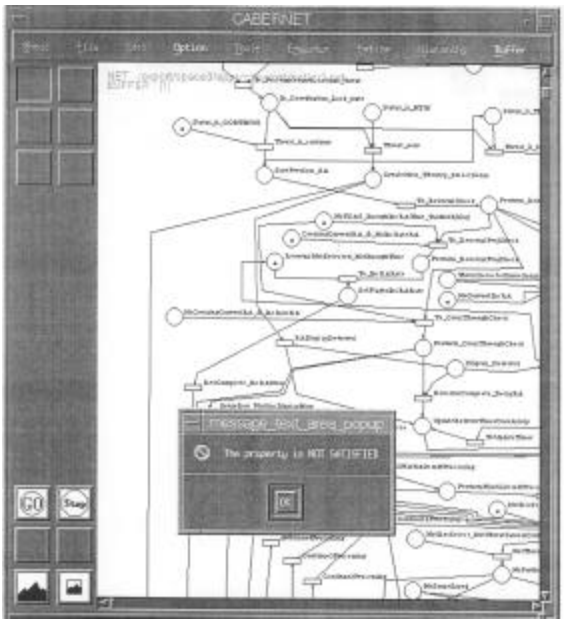

Fig. 6. Sample Output from Cabernet.

The automatic verification was very successful, since it allowed to identify several flaws in the TCAS specification. The results showed, however, that a significant knowledge of Petri nets (timed environment relationship nets, in this case) is needed to build a meaningful TCAS model. In order to enhance usability, it is useful to support formalization by means of established engineering practice. One such practice emerging recently, with supporting software tools, is the use of a Unified Modeling Language (UML).

6. VERIFYING TIMING REQUIREMENTS USING UML

This section presents a method of enhancing UML by dealing with timing requirements via statecharts. We propose to combine traditional engineering tools, such as those relying on the UML notation, with formal methods tools, such as model checkers. Combining both types of tools into a single integrated system via a common user interface, to express and verify timing properties, leads to an architecture of a verification system (Al-Daraiseh et al. 2001) as in Fig. 7.

Two tools, a typical design tool based on UML, and a typical model checker based on a formal method, are interfaced to a user via a GUI. For expressing timing requirements, UML statecharts are converted to Extended Timed Graphs (XTGs), which in turn are converted to textual representation and analyzed by a model checker. A new software tool has been developed to ease the interfacing of models built in XTG formal language to other tools.

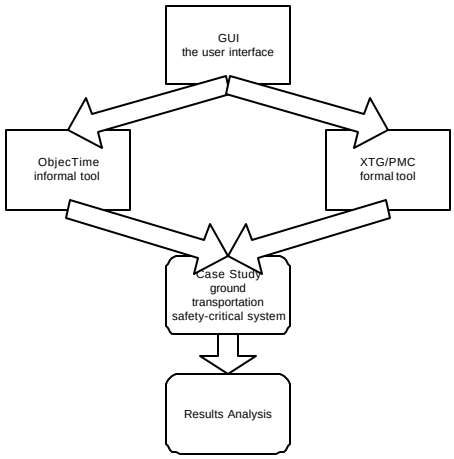

Fig. 7. Architecture of the Integrated System.

6.1 Integrated Methodology

Both statecharts and XTG's are visual tools to model the behavior of software. Since UML has no means to express real-time properties, we propose a method to convert the UML statecharts to XTGs, so that statechart models can be model checked using the PMC model checker (van Katwijk et al. 2000). Converting a UML model expressed in statecharts, to XTG graphs, requires thorough understanding of their mutual correspondence.

XTG is a new engineering notation for describing real-time systems based on timed automata (Ammerlaan et al. 1998). It provides a simple representation for high-level specification languages and is a suitable notation for those languages that allow extensive modeling of data, having a maximal progress semantics, and modeling interprocess communication by value passing through data channels.

A UML statechart consists of states and transitions. A state is an ontological condition that persists for a significant period of time, is distinguishable from other such conditions, and is disjoint from them. A distinguishable state means that it differs from other states in the events it accepts, the transitions it takes as a result of accepting those events, or the actions it performs. A transition is a response to an event that causes a change in state.

A simple statechart diagram is shown in Fig 8. In the IDLE state nothing is happening and the timer is not counting down, waiting for the start command "StartCmd". The StartCmd carries a single value, which is the starting time the timer shall count down.

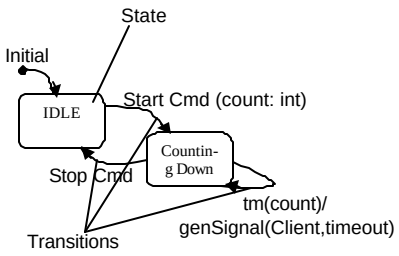

Fig. 8. Simple Statechart Representing a Counter.

When this event occurs the system transfers to the Counting Down state and the timer starts counting down again. There is also a timeout signal "tm". When the timeout occurs the "tm" transition is taken and the action from its action list will be executed. The actions are shown after the slash "/". In this case the action is to send a signal to a client object. Actions in UML are usually short but they run until completion before the transition completes. After that, the counting state is re-entered, and the counting starts again until the Stop Cmd event occurs. Then the system transfers to IDLE waiting for the Start.

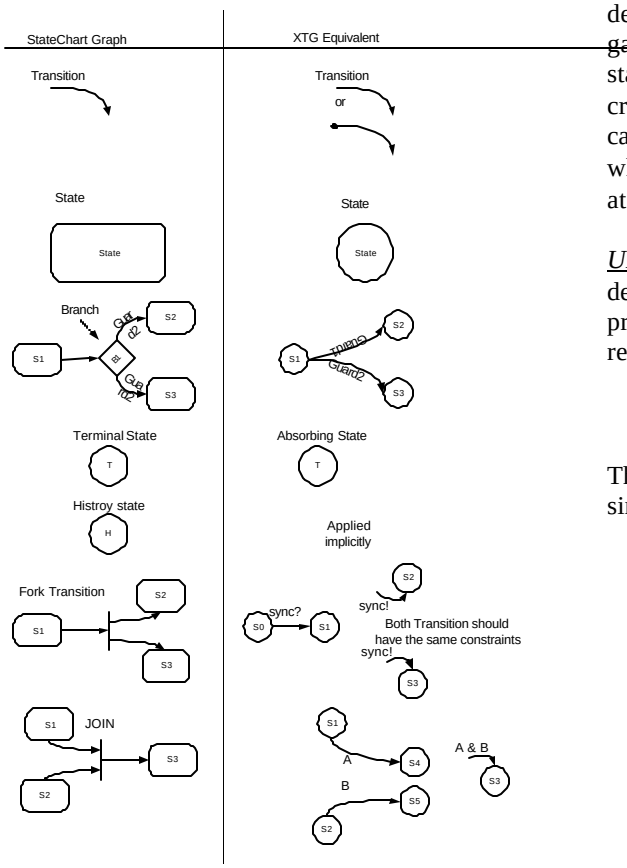

Fig. 9. Statechart to XTG Graphs Conversion.

To convert statecharts to XTG graphs, we developed an equivalence table (partially presented in Fig. 9), that contains the statechart symbols and the XTG equivalent symbols. As shown in Fig. 9 the state symbols are equivalent but their shapes are different. Transitions in both cases are the same. According to the table, the equivalent XTG graph for the Counter example is shown in Fig. 10.

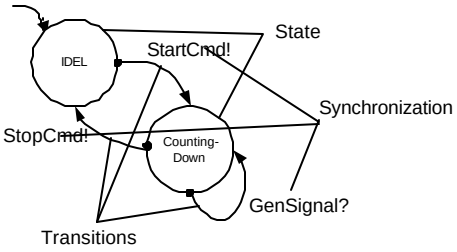

Fig. 10. XTG Graph Equivalent to Fig. 8.

6.2 Railroad Crossing Case Study

To check the usefulness of the concept of model conversion and the applicability of the tool developed for this purpose, we applied the entire procedure to the railroad crossing system. The system consists of three parts: a set of sensors detecting the passing trains, a gate controller, and a gate. The sensors process can be in one of three states namely: far from the crossing, T0, near the crossing, T1, and in the crossing, T2. A function called g(t) is defined to represent the gate formally, where output of this function g(t)∈ [0,90] means that at 0 the gate is closed and at 90 the gate is open.

UML Design. The behavior of the system is described with UML in Fig. 11. The safety and utility properties are expressed in temporal logic (z represents clock) as follows:

1. AG(train@in => gate@closed)
2. AG(gate@closed => (z=0).
   AF (gate@open and z <=5))

The timing properties are missing on the diagram, since they cannot be expressed in UML.

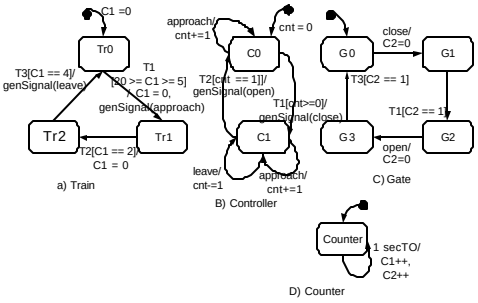

Fig. 11. UML Model of System Behavior.

As shown in this figure, the UML representation has to include four processes instead of three, as suggested previously, since UML doesn't have the clock variable. The sensors process can be in one of three states:

- Far from the crossing, Tr0
- Near the crossing, Tr1
- In the crossing, Tr2.

The railroad crossing example was implemented in ObjecTime tool as shown in Fig. 12, using four actors: TrainActor, GateActor, ControllerActor and CounterActor. ObjecTime behavioral diagram is similar to UML statecharts. The behavior of the GateActor is simulated internally. The principle of operation is the same as described in the UML design.

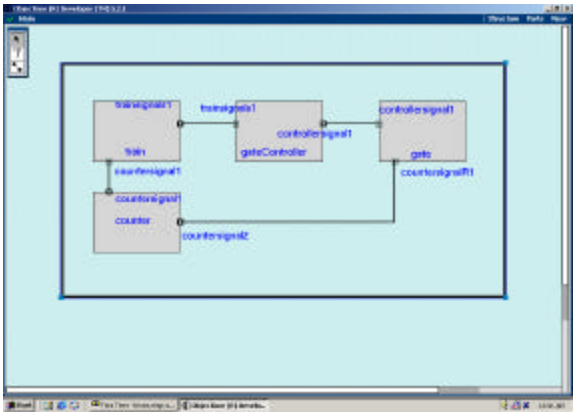

Fig. 12. Railroad Crossing ObjecTime Model.

With ObjecTime model simulation, we could only check sequencing and a limited timing behavior via animation, but there was no proof of the safety constraints since the tool does not have this capability.

XTG Design. Next the UML description is translated into XTG's. The gate has four possible states. Initially it is open (G0). When receiving a *close* signal, it transfers to the closing state (G1). One time unit is allowed to close the gate, enforced by [C2==1] clock constraint.

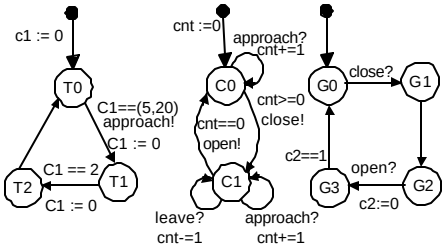

Fig. 13. Three XTG Processes Representing the System: Sensors, Controller and Gate.

This description is sufficient to use the XTGConverter tool for producing an XTG code, form the XTG graphs, and prepare for automatic model checking. Results for railroad crossing are presented in (van Katwijk et al. 2000, and Al-Daraiseh et al. 2001). A portion of the generated XTG code for the sensors process is shown in Fig.14.

```
-- this is the RailRoadCrossing System
system RailRoadCrossing
. . .
processes
        Sensors         Train1 ;
        Gate            Gate1 ;
        GateCont        GateCont1;
composition
        Train1 | Gate1 | GateCont1

graph Sensors
-- Local variable assignment added here
state
-- clocks and variables here
clock C1:= 0;
ports
--all the ports are added here
        out approach
        out leave
--initial state comes after init keyword
init
        T0
locations
. . .
T0
{
when C1 == (5,20)
sync approach!
do  C1 := 0;
goto    T1
}
```

Fig. 14. XTG Code for Model Checking.

Finally, after running the generated code through the PMC model checker, the verification results are obtained. The properties to be verified in the railroad crossing system are:

1. `AG(train@in => gate@closed)`
2. `AG(gate@closed => (z=0).`
   `AF(gate@open and z <=5))`

This means that, at any time, if there are trains in the crossing the gate should be closed, and at any time if the gate is closed, it should open in the future.

7. CONCLUSION

Modern safety related systems can be so complex and so software dependent, that dealing with the software production process becomes indispensable to achieve the required level of trustworthiness in software.

Usual verification methods, such as simulation and informal prototyping, are dealing only with the products, not with the processes, and may work well for checking individual, safety related or timing properties, but not for the verification of the correctness of development processes. On the other hand, the use of diverse software for safety critical

systems is process-related, basing on the assumption of dissimilar development methodologies. Also the evaluation of commercial off-the-shelf software to be re-used in a safety-critical context requires a careful consideration of the original development process.

On the whole, the approaches described in this article are complementary with respect to their suitability to address different problems at different levels of formalism. An adequate combination is considered to be promising. One way to proceed is to incorporate formal techniques into engineering practice and support it by a combined use of automatic tools, so that formal verification of safety properties may become more automated and better accessible to engineers and software developers

REFERENCES

Al-Daraiseh A., J. Zalewski, H. Toetenel (2001), Software Verification in Ground Transportation Systems, *Proc. SCI2001, 5th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, Fla., July 22-25

Ammerlaan M., R.L. Spelberg, H. Toetenel (1998), XTG – An Engineering Approach to Modelling and Analysis of Real-Time Systems, *Proc. 10th Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, pp. 88-97

Anderson E., J. van Katwijk, J. Zalewski (1999), New Method of Improving Software Safety in Mission Critical Real-Time Systems, *Proc. 17th Int'l System Safety Conference*, Orlando, FL, August 16-21, System Safety Society, Unionville, Va., pp. 587-596,

Bishop P., ed. (1990), *Dependability of Critical Computer Systems: Guidelines. Techniques Directory*. Elsevier Applied Science, London

Eckhardt Q.E. Jr., L. D. Lee (1985), A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors, *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 12, pp. 1511-1517

Ehrenberger W. (2001a), Software Diversity: Some Considerations on Failure Dependency, *Proc. SCI2001, 5th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, Fla., July 22-25

Ehrenberger W. (2001b), Software-Verifikation, Hanser Verlag, Munich

Ehrenberger W., F. Saglietti (1993), Architecture and Safety Qualification of Large Software Systems, *Proc. ESREL'93, European Safety and Reliability Conference*, Munich, Germany, May 12-14, Elsevier, Amsterdam, pp. 985-999

Hilburn T., J. Zalewski (1996), Real-Time Safety Critical Systems: An Overview, *Proc. 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, Elsevier, Oxford, pp. 127-138

van Katwijk J., H. Toetenel, A.E.K. Sahraoui, E. Anderson, J. Zalewski (2000), Specification and Verification of a Safety Shell with Statecharts and Extended Timed Graphs, *Proc. SAFECOMP 2000, 19th Int'l Conf. On Computer Safety, Reliability and Security*, Springer-Verlag, Berlin, pp. 37-52

Kornecki A., B. Nasah, J. Zalewski (1998), TCAS Safety Analysis Using Timed Environment-Relation Petri Nets, *Proc. ISSRE'98, Int'l Symposium on Software Reliability Engineering*, Paderborn, Germany, November 4-7

Leveson N. (1996), *Safeware: System Safety and Computers*, Addison-Wesley, Reading, Mass.

Littlewood B., D. R. Miller (1987), A Conceptual Model of Multi-Version Software, *Proc. FTCS-17, Int'l Symp. On Fault-Tolerant Computing*, IEEE Computer Society Press, pp. 170-175

Mojdehrakhsh R., W.T. Tsai, S. Kirani, L. Elliott (1994), Retrofitting Software Safety in an Implantable Medical Device, *IEEE Software*, Vol. 11, No. 1, pp. 41-50

Rushby J. (1994), Critical System Properties: Survey and Taxonomy, *Reliability Engineering and System Safety*, Vol. 43, pp. 189-219

Saglietti F., W. Ehrenberger, M. Kersken (1992), *Software Diversität für Steuerungen mit Sicherheitsverantwortung*, Report BAU-Forschungsbericht FB 664, Bundesanstalt für Arbeitsschutz, Dortmund

Saglietti F. (1998), Integration of Logical and Physical Properties of Embedded Systems by Use of Timed Petri Nets, *Proc. SAFECOMP'98, 17th Int'l Conf. On Computer Safety, Reliability and Security*, Springer-Verlag, Berlin, pp. 319-328

Saglietti F. (2000a), Evaluation of Pre-developed Software for Use in Safety-Critical Systems, *Proc. EUROMICRO'2000, 26th Euromicro Conference on Software Process and Product Improvement*, IEEE Computer Society Press, Vol. 2, pp. 193-199

Saglietti F. (2000b), Statistical Significance of Expert Judgement for Ultrahigh Software Reliability Demands, *Proc. 5th Int'l Conference on Probabilistic Safety Assessment and Management*, Osaka, Japan, Nov. 27-Dec. 1, Universal Academy Press, Tokyo

Saglietti F. (2001), Criticality and Sensitivity Analysis for Off-the-Shelf Components in Safety-Relevant Systems, *Proc. SCI2001, 5th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, Fla., July 22-25

Sahraoui A.E.K., E. Anderson, J. van Katwijk, J. Zalewski (2000), Formal Specification of a Safety Shell in Real-Time Control Practice, *Proc. WRTP'2000, 25th IFAC Workshop on Real-Time Programming*, Elsevier, Oxford, pp. 117-123