Methodology and Experience for Designing Safety-Related Systems in IEC 61508

Zhe Chen
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
29 Yudao Street, 210016 Nanjing, China
Email: zhechen@nuaa.edu.cn

Gilles Motet
LAAS-CNRS, INSA
Université de Toulouse
135 Avenue de Rangueil, 31077 Toulouse, France
Email: gilles.motet@insa-toulouse.fr

***Abstract*—The international standard IEC 61508 provides a generic process for electrical, electronic, or programmable electronic (E/E/PE) safety-related systems (SRS) to achieve an acceptable level of functional safety. This paper first proposes the concept of *functional validity* of SRS, based on our observation on two important problems that occur in industrial practice, i.e., the rightness of overall and allocated safety requirements and the lack of technical methodologies for validating SRS. *Functional validity* means whether the safety functions realized by SRS can really prevent accidents and recover the system from hazardous states, provided the expected safety integrity level is reached. Then this paper proposes a generic technical methodology to achieve the functional validity of SRS, and summarizes industrial experiences in designing functionally valid SRS. A concrete example is used to illustrate the proposed methodology.**

***Keywords*-safety-related system; IEC 61508; functional validity; verification; model checking; formal method; SPIN**

I. Safety-Related Systems and Functional Validity

The international standard IEC 61508 [1][2][3] provides a generic process for electrical, electronic, or programmable electronic (E/E/PE) safety-related systems to achieve an acceptable level of functional safety. The principles of IEC 61508 have been recognized as fundamental to modern safety management [2], thus have gained a widespread acceptance and been used in practice in many countries and industry sectors [4].

Like other safety standards (e.g., DO-178B [5]), IEC 61508 gives recommendations on best practices such as planning, documentation, verification, safety assessment, rather than concrete technical solutions. Thus, it is a generic standard for the safety management throughout the life-cycle, rather than a system development standard. More sector-specific and application-specific standards can be derived based on the standard, such as IEC 61511 for process industry [6], IEC 62061 for machinery industry, IEC 61513 for nuclear plants, EN 50126 for European railway, and ISO 26262 for automotive safety.

As shown in Fig. 1, the first premise of the standard is that there is an equipment intended to provide a function, and a system which controls it. The equipment is called an *Equipment Under Control* (EUC). The *Control System* (CS) may be integrated with or remote from the EUC. A fundamental tenet of the standard is that, even if the EUC and the CS are reliable, they are not necessarily safe. This is true for numerous systems implementing hazardous specification. They may pose risks of misdirected energy which result in accidents.

The second premise is that *Safety-Related Systems* (SRS) are provided to implement the expected *safety requirements*, which are specified to reduce the risks and achieve functional safety for the EUC. The SRS may be placed within or separated from the CS. In principle, their separation is preferred.

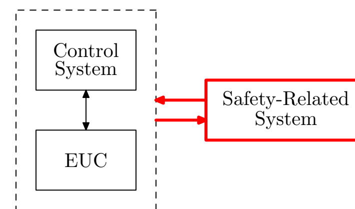

Figure 1. The Architecture of Systems with SRS

An SRS may comprise subsystems such as sensors, logic controllers, communication channels connected to the CS, and actuators. Usually, an SRS may receive two types of input: the values of safety-related variables monitored by its sensors, and the messages sent by the CS. Then the SRS executes computation to decide whether the system is in a safe state. According to the result, it may actualize safety functions through two types of output: directly sending commands to its actuators, or sending a message to the CS.

Let us consider two important problems that occur in industrial practice.

The first one questions the *rightness* of overall and allocated safety requirements. According to IEC 61508, safety requirements consist of two parts, *safety functions* and associated *safety integrity levels* (SIL). The two elements are of the same importance in practice, because the safety functions determine the maximum theoretically possible risk reduction [7]. However, the standard focuses more on the realization of integrity requirements rather than function requirements. As a result, the standard indicates only that

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-149-6

the product is of a given reliable integrity, but not that it implements the *right* safety requirements.

Second, the standard does not prescribe exactly *how* the verification of safety functions of an SRS could technically be done. On one hand, the standard calls for avoiding faults in the design phase, since the ALARP principle (As Low As Reasonably Practicable) is adopted for determining tolerability of risk. Indeed, *systematic faults* are often introduced during the specification and design phases. Unlike random hardware failures, the likelihood of systematic failures cannot easily be estimated. On the other hand, the standard only recommends a *process* and a list of techniques and measures during the design phase to avoid the introduction of systematic faults, such as computer-aided design, formal methods (e.g., temporal logic), assertion programming, recovery (see parts 2, 3 of [1]). The detailed use of these techniques is left to the designer.

As a result, the problem of functional validity arises. ***Functional validity*** **means whether the** ***safety functions*** **realized by SRS can really prevent accidents and recover the system from hazardous states**, provided the expected safety integrity level is reached. People are searching for a generic technical methodology to achieve functional validity.

In fact, with the introduction of SRS, it becomes much harder to ensure the safety of the overall system, due to complex interactions and the resulting huge state space. Unlike the case of a single CS, human is no longer capable to analyze manually and clearly the behaviors of the overall system. Therefore, we shall expect a computer-aided method.

This paper proposes such a generic technical methodology (or a framework) for designing *functionally valid* SRS. To the best of our knowledge, we are the first to consider the technical solution to functional validity of SRS in the literature. We focus on the systems that operate on demand (i.e., discrete event). The methodology is based on computer-aided design in association with automated verification tools (e.g., SPIN). In Section II, we present a concrete example illustrating the application of the proposed methodology, which is formally discussed in Section III. We discuss related work in Section IV, then conclude in Section V.

II. Example: Chemical Reactor

As an example, consider an accident occurred in a batch chemical reactor in England [8][9]. Figure 2 shows the design of the system. The computer, which served as a control system, controlled the flow of catalyst into the reactor and the flow of water for cooling off the reaction, by manipulating the valves. Additionally, the computer received sensor inputs indicating the status of the system. The designers were told that if an abnormal signal occurred in the plant, they were to leave all controlled variables as they were and to sound an alarm.

On one occasion, the control system received an abnormal signal indicating a low oil level in a gearbox, and reacted as the functional requirements specified, that is, sounded an alarm and maintained all the variables with their present condition. Unfortunately, a catalyst had just been added into the reactor, but the control system had not yet opened the flow of cooling water. As a result, the reactor overheated, the relief valve lifted and the contents of the reactor were discharged into the atmosphere.

We believe that an SRS could be used to avoid the accident. Figure 3 shows the role of the SRS in the overall system. It receives signals from additional sensors, and communicates with the CS. The key issue is how to specify and design the SRS and prove its *functional validity*, i.e., the SRS is really efficient in the hazardous context. We illustrate a methodology based on computer-aided design in association with the SPIN model checker.

The SPIN (Simple Promela INterpreter) model checker is an automated tool for verifying the correctness of asynchronous distributed software models [10][11]. Systems and correctness properties to be verified are both described in Promela (Process Meta Language). SPIN also supports Linear Temporal Logic (LTL) formulas, which are converted into never claims written in Promela for verification. Besides a checker, SPIN can also operate as a simulator by executing the model following one possible execution trace.

We will illustrate two main steps of the methodology: modeling the CS, and modeling the SRS.

Modeling Control Systems. The first step is to analyze the behaviors of the CS by modeling it using Promela. Listing 1 shows the model. The CS scans the status of the reactor, and then manipulates the valves according to the status.

Listing 1. The Promela Program for Reactor Control System

```
1  #define sa ((abnorm && !cata)||(abnorm && cata && water))
2  #define fcon status==nocata || status==encata
3  mtype = {abnormal, nocata, encata, nowater, enwater};
4  mtype status = nocata; /* status of the reactor */
5  bool cata   = false;   /* whether catalyst flow is open */
6  bool water  = false;   /* whether water flow is open */
7  bool abnorm = false; /* whether abnormal signal occured */
8
9  /* random simulation of scanning the status */
10 inline scan() {
11 if
12   :: true -> status = abnormal;
13   :: cata  == false -> status = nocata;
14   :: cata  == true  -> status = encata;
15   :: water == false -> status = nowater;
16   :: water == true  -> status = enwater;
17 fi;
18 }
19
20 /* possible actions of the system */
21 inline opencata()  {cata =true; printf("open cata -> "); }
22 inline closecata() {cata =false;printf("close cata -> ");}
23 inline openwater() {water=true; printf("open water -> ");}
24 inline closewater(){water=false;printf("close water ->");}
25 inline alarm() { abnorm = true; printf("alarm -> "); }
26 inline ending(){ printf("ending -> "); }
27
28 active proctype controlsystem(){
29 /*Initial actions omitted*/
```

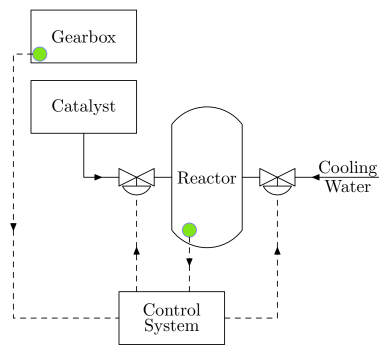

Figure 2. Reactor Control System

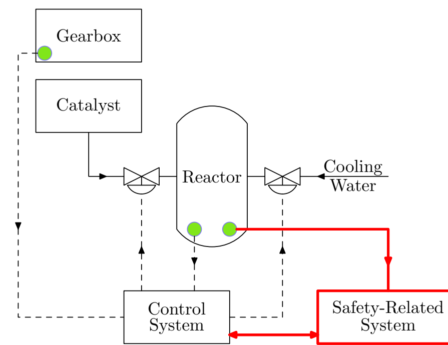

Figure 3. Reactor Control System with SRS

```
30 do
31 :: scan();
32    if
33    :: status == abnormal -> alarm(); goto END;
34    :: else -> if
35       :: status==nocata  && cata ==false->opencata();
36       :: status==encata  && cata ==true ->closecata();
37       :: status==nowater && water==false->openwater();
38       :: status==enwater && water==true ->closewater();
39       :: else -> skip;
40       fi;
41    fi;
42 od;
43 END: ending();
44 assert(sa);
45 }
```

Lines 1-2 define macros for specifying correctness properties.

Lines 3-7 define the variables. Note that `status` denotes the detection of events (e.g., abnormal signal, no catalyst or enough catalyst in the reactor), and controls the flow of the computation, while the other three variables save the state information. Note that abnormal signal may be caused by several events. For example, low oil level is one of these events.

Lines 10-18 simulate the input events to the CS. In the verification mode, SPIN can only treat closed system without users' input. Therefore, the status of reactor is generated in a random manner, i.e., the `if` statement (lines 11-17) chooses randomly one of the alternative statements whose condition holds. Note that this matches exactly what happens in practice, since the status of the reactor is determined nondeterministically by the reaction and the environment, not the user.

Lines 21-26 define the primitive actions of the CS. To obtain an efficient model for verification, the real codes for manipulating physic equipments are replaced by `printf` statements, which could be used for observing the execution traces of the CS.

Lines 28-45 specify the model of the CS. The non-critical codes (irrelevant to the concerned property, e.g., the code for initiation) are omitted or simplified to produce an efficient model. In line 44, we check whether the system satisfies the safety assertion `sa` (cf. line 1), which means when an abnormal signal occurs, either the flow of catalyst must be closed, or the flow of water must be also open if the flow of catalyst is open. The violation of this assertion may result in the mentioned accident.

In order to see whether the model describes exactly the behaviors of the CS, we run the model through the simulation mode of SPIN. One of the outputs is as follows.

```
1 $ spin reactor.pml      # Linux command
2 open cata -> alarm -> ending ->
3 spin: line  44 "reactor.pml", Error: assertion violated
```

The execution trace shows that the alarm is sounded after opening the flow of catalyst, then the safety assertion is violated. This trace characterizes exactly what happened in the accident. It is worth noting that, due to the uncertainty of the value of `status` (cf. lines 11-17), the assertion may be satisfied in another run. This is the reason why the plant had functioned well before the accident. As a result, in the simulation mode, it is very possible that an existing fault will not be detected after numerous runs.

To systematically check all the state space, we use the verification mode, which needs correctness properties specifying hazard-free situation.

One possibility is to use assertions, such as what we did in line 44. Note that an assertion can only check the state at a single position in the execution (i.e., line 44), not the remainder positions. The SPIN checks the assertion in the verification mode as follows.

```
1 $ spin -a reactor.pml
2 $ gcc pan.c
3 $ ./a.out
4 State-vector 16 byte, depth reached 12, errors: 1
5        21 states, stored
6         0 states, matched
7        21 transitions (= stored+matched)
```

After examing 21 states, SPIN detected the unsafe state. We can simulate the counterexample by tracing simulation.

```
1 $ spin -t reactor.pml
2 open cata -> alarm -> ending ->
3 spin: line  44 "reactor.pml", Error: assertion violated
```

The result shows that the unsafe state can really be reached.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-149-6

Another alternative for specifying correctness property is LTL formula. For this example, the formula is $\phi \stackrel{\text{def}}{=} \mathbf{G}(cata \rightarrow \mathbf{F}water)$, where $\mathbf{G}$ means *globally* and $\mathbf{F}$ means *future* [12]. It means whenever the flow of catalyst is opened, the system will have the chance to open the flow of water in the future. Of course, this is based on a reasonable assumption that the status will eventually be `nowater` when there is not enough water in the reactor (i.e., there is a reliable sensor). This assumption is expressed by the fairness condition $\mathbf{GF}!fcon$. The SPIN checks the LTL formula in the verification mode as follows (`[]` denotes $\mathbf{G}$, `<>` denotes $\mathbf{F}$).

```
$ spin -a -f '![](cata -> <>water)&&[]<>!fcon' reactor.pml
$ gcc pan.c
$ ./a.out -A -a #disable assertion, check stutter-
      invariant
State-vector 20 byte, depth reached 29, errors: 1
       23 states, stored
        1 states, matched
       24 transitions (= stored+matched)
```

After examing 23 states, SPIN detected the unsafe state. We can simulate the counterexample by tracing simulation.

```
$ spin -t reactor.pml
open cata -> alarm -> ending ->
spin: trail ends after 30 steps
```

Since we have already known that the CS may cause a hazardous state, we must make sure that the specified properties can detect the hazardous state, or else the properties are not right. Then we introduce an SRS to control potential hazardous executions.

Modeling Safety-Related Systems. The second step is to construct the model of the SRS. We reuse the established model of the CS, and derive an accurate model describing the behaviors of the SRS. Listing 2 shows the models. The model of the SRS receives messages sent by the CS, computes whether the system is safe, and sends a message back to the CS.

Listing 2. Reactor Control System with SRS in Promela

```
#define sa ((abnorm && !cata)||(abnorm && cata && water))
#define fcon status==nocata || status==encata
mtype = {abnormal, nocata, encata, nowater, enwater};
mtype status = nocata; /* status of the reactor */
bool cata   = false;  /* whether catalyst flow is open*/
bool water  = false;  /* whether water flow is open */
bool abnorm = false; /* whether abnormal signal occured*/

/*define safety-related variables, messages structure*/
typedef SRV { bool _water; }
typedef MSG { bool _water; bool _abnorm; bool _res; }
chan ch = [0] of { MSG };      /* message channel */

/*random simulation of scanning the status*/
inline scan() {
if
  :: true -> status = abnormal;
  :: cata  == false -> status = nocata;
  :: cata  == true  -> status = encata;
  :: water == false -> status = nowater;
  :: water == true  -> status = enwater;
fi;
}

/* possible actions of the system */
inline opencata() {cata=true; printf("open cata -> ");}
inline closecata(){cata=false;printf("close cata-> ");}
inline openwater()
  { water = true;  cs_2srs(); printf("open water -> "); }
inline closewater()
  { water = false; cs_2srs(); printf("close water -> ");}
inline alarm(){abnorm=true;cs_2srs();printf("alarm ->");}
inline ending() { printf("ending -> "); }
inline cs_epro()
    { printf("error processing (");
      printf("water opened");
      water = true;
      printf(") -> "); }

/* communication between CS and SRS */
inline cs_2srs() {
if
  :: abnorm == true  -> msg._abnorm = true;
  :: abnorm == false -> msg._abnorm = false;
fi;
if
  :: water == true  -> msg._water = true;
  :: water == false -> msg._water = false;
fi;
msg._res = true;
ch ! msg;
ch ? msg;
if
  :: msg._res == false -> cs_epro();
  :: else -> skip;
fi;
}

active proctype controlsystem(){
/*Initial actions omitted*/
MSG msg;
do
:: scan();
   if
   :: status == abnormal -> alarm(); goto END;
   :: else -> if
      :: status==nocata  && cata ==false ->opencata();
      :: status==encata  && cata ==true  ->closecata();
      :: status==nowater && water==false ->openwater();
      :: status==enwater && water==true  ->closewater();
      :: else -> skip;
      fi;
   fi;
od;
END: ending();
assert(sa);
}

/***** The Safety-related System *****/
/*random simulation of scanning the values of variables*/
inline srs_scan() {
if
  :: srv._water = true;
  :: srv._water = false;
fi;
}

/* compute whether the system is safe */
inline srs_compute() {
if
  :: msg._abnorm == true && msg._water == false ->
                    msg._res = false;
  :: msg._abnorm == true && srv._water == false ->
                    msg._res = false;
  :: else -> msg._res = true;
fi
}

active proctype srs(){
/*Initial actions omitted*/
MSG msg;
SRV srv;
do
  :: true ->
```

```
105 endsrs: ch ? msg;
106         srs_scan();
107         srs_compute();
108         ch ! msg;
109 od;
110 }
```

Lines 10-11 define the inputs to the SRS. The type `SRV` defines a set of safety-related variables, whose values could be obtained from additional sensors outside the CS and managed by the SRS. For this example, `SRV` monitors only whether the water flow is open. The type `MSG` defines the structure of the messages communicated between the CS and the SRS. Line 12 defines a rendezvous channel for the message.

In lines 28-32, for each primitive action that modifies the values of the variables monitored by the SRS, the communication between the CS and the SRS is inserted. The communication module (lines 41-57) reads information needed, and sends a message to the SRS, then receives a response. The module analyzes the result in the returned message. If it indicates an unsafe state, the system calls the error processing module to recover from the hazardous state.

The error processing module (lines 34-38) uses the information in the returned message to decide the actions. The process could change the values of certain variables (e.g., line 37) after manipulating physic equipments (e.g., in line 36, `printf` statement abstracts the action of opening the valve). Note that more information could be contained in the returned message in more complex systems (i.e., not only a boolean result), in order to provide sufficient information for the error processing module to analyze the current state.

Lines 99-110 define the model of the SRS. It waits for the message sent by the CS, then scans the values of the safety-related variables (lines 81-86), and computes whether the system is safe using the message and the safety-related variables (lines 89-97). Finally the SRS sends a response to the CS. Note that the computation in `srs_scan` and `srs_compute` could be different in various systems. Embedded C code could be used to implement more complex functions. Anyway, we use the same methodology and framework.

In order to see whether the model characterizes exactly the behaviors of the SRS, we run the model through the simulation mode of SPIN. One of the outputs is as follows.

```
1 open cata -> error processing ( water opened ) -> alarm ->
      ending ->
```

The execution trace shows that the safety assertion is not violated, which is exactly what we expect to avoid the mentioned accident.

Then we check the assertion in the verification mode, and no error is found.

```
1 State-vector 40 byte, depth reached 208, errors: 0
2       409 states, stored
3        57 states, matched
4       466 transitions (= stored+matched)
```

We may also check the LTL formula in the verification mode.

```
1 State-vector 44 byte, depth reached 401, errors: 0
2       707 states, stored (979 visited)
3       658 states, matched
4      1637 transitions (= visited+matched)
```

Zero errors mean that the assertion holds and the LTL property always holds in the execution, i.e., no unsafe state could be reached. Therefore, we conclude that the established model of the SRS can successfully avoid the accident, i.e., a "functionally valid" SRS. Note that, if we use another computation in `srs_compute`, the SRS may be not functionally valid (e.g., always let `msg._res` be `true`). That is the reason why we need such a methodology to ensure functional validity.

It is worth noting that the combination of the CS and the SRS has 707 states and 1637 transitions (more complex the overall system, larger the state space). Human is not able to analyze the correctness of such a complicated system. As a result, computer-aided design may be the only choice for developing functionally valid SRS.

III. Methodology for Designing Functionally Valid SRS

In this section, we propose the generic methodology for designing functionally valid safety-related systems. As we mentioned, the state space of the overall system including the CS and the SRS is much larger than the one of a single CS or a single SRS. As a result, manual analysis and design of the SRS are never trustworthy and always error-prone. This methodology uses computer-aided design in association with automated verification tools, thus can improve our confidence on the functional validity of the design of SRS.

We try to list exhaustively all the key issues that we know in the designing process, in order to guide the practice. Due to the wide application of the standard and SRS, the reader may encounter different situation in various projects and industry sectors. Therefore, some necessary adaptations should be made in detail for a specific project.

Generally, there are three steps for developing a functionally valid SRS: modeling the CS, modeling the SRS and implementing the SRS. This paper focuses on the first two steps (i.e., the design process) which lead to a functionally valid design of the SRS, although it is worth noting that faults may be also introduced in the implementation process.

Modeling Control Systems. The first step is to construct the model of the CS. The results of this step are a Promela program for the CS and the correctness properties. We list some key issues as follows.

(1) The first task is to derive *accurate* and *efficient* abstraction of the behaviors of the CS. The criteria for judging the quality of the model are mainly *accuracy* and *efficiency*. Accuracy means that the model behaves exactly

like the real CS, while efficiency means that the model is smart enough, e.g., the program should use as few variables, statements and memory as possible. Some typical issues are the following ones:

(a) Define variables, both for computation and checking. Some variables are used for computation, that is, implementing control flow, behaviors and semantics of the system. Some variables are used for representing the state of the system, so they do not contribute to the functionality of the system. They are only used in the correctness properties to check the property of the indicated state.

It is worth noting that the size of variables must be carefully defined. The principle is “as small as possible”. The model checker will produce a huge state space, of which each state contains all the defined variables. As a result, the restriction from `int` to `byte` will save considerable memory when checking.

(b) Simulate the random inputs to the CS. In the verification mode, the Promela model is a closed system. In other words, it cannot receive users’ inputs. As a result, we must generate the inputs in the program.

The best way is to generate the inputs randomly, i.e., nondeterministically choose one member from the set of all possible values. The advantage of this method is that it can simulate the uncertainty of the inputs, that is, we do not know when a specific input will occur. For example, consider the sensors’ signal which is determined by the environment.

(c) Simplify reasonably the computation of the CS. Due to the size and complexity of the real system, an automated model checker may not even be able to produce the result in an acceptable instant. Obviously, the huge size contributes to a huge number of states, and the complexity contributes to a huge number of transitions. As a result, the size of the state space may be much larger than the memory, then the model checker will fail to accomplish the verification.

One solution is to provide a more coarse abstraction of the CS. That is, we omit some non-critical computations, e.g., the initiation of the system. Furthermore, some manipulations of physical equipments can be expressed with only a `printf` statement, which does not increase the size of the state space.

Another solution is to decompose the system into several parts, and check these parts one by one. When checking one single part, we make the assumption that the other parts are correct. It is worth noting that the decomposition is relevant to the properties to check. That is, we must put all the functions relevant to a certain property into the same part, when checking the property.

(d) Use embedded C codes if necessary. Due to the complexity of embedded C codes and the lack of syntax checking, they are mainly used for automated model extraction in SPIN. However, the strong expressive power of C code is anyway a tempting feature. Thus the recommendation is made only “if necessary”.

(e) Simplify or eliminate the codes for controlling equipments. Usually we assume that the codes for implementing primitive manipulations are correct, e.g., opening the flow of water.

(2) The second task is to derive correctness properties, i.e., assertions and LTL formulas.

(a) Assertions are used to check the property at a specific position in the program. Obviously, the expressive power is limited. Thus, if we want to check a property over all the state space, we must use LTL formulas.

(b) LTL formulas are used to check all the states in the system. Obviously, LTL formulas are more powerful. However, it is also worth noting that LTL formulas can considerably largen the state space. Thus we suggest to use them only when assertions are not able to express a property.

(3) Some Experiences.

(a) Check the exactitude of the established model, by using simulation mode. The `printf` statement can be used to output information about the state of the CS. Simulating the model for sufficient many times can show whether the model works as expected. Note that the criteria “sufficient many times” is due to the uncertainty of the inputs.

(b) Check the exactitude of the correctness properties, by using verification mode. If we aim at creating or modifying an SRS for identified risks, we must make sure that the correctness properties can detect the error caused by the identified risks.

Modeling Safety-Related Systems. The second task is to construct the model of the SRS. The results of this step are a Promela program for the SRS and the codes for communication and error processing in the modified model of the CS. We list some key issues as follows.

(1) The premise is that we have the model of the CS and reuse it, e.g., the results of the last step.

(2) The first task is to derive *accurate* and *efficient* abstraction of the behaviors of the SRS. Some typical issues are the following ones:

(a) Define the scope of input and output of the SRS. There are two types of input: the message sent by the CS and the values of safety-related variables sent by the sensors. The SRS may use only one of the two types, or both of them. There are two types of output: the message sent to the CS and the direct manipulation of the equipment. Also, the SRS may use only one of the two types, or both of them.

(b) Define safety-related variables. A *safety-related variable*, which saves a value sent by sensors, is used to collect additional information which is beyond the scope of the CS, or to collect a specific piece of information in the scope of the CS for increasing the reliability of the information. Note that more safety-related variables means higher cost of implementation.

(c) Choose the type of communication between the CS and the SRS. The SPIN model checker supports two types of communications: rendezvous and buffered communication.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-149-6

In order to process the demand from the CS to the SRS as soon as possible, and also provide information from the SRS to the CS to decide the next action, we usually choose the rendezvous communication.

(d) Define the structure of message. This depends on the information needed by the CS and the SRS. The values of the variables monitored by the SRS should be sent from the CS to the SRS. The message should also contain all the necessary information needed by the CS to deal with risks. In the simplest case, it is a Boolean result, indicating the system is safe or not. However, generally, the CS need more information to determine why the system is not safe, then it can activate corresponding error processing functions.

(e) Define message channels. Usually, one channel is enough for the communication between the CS and the SRS.

(f) Simulate the scanning of the values of safety-related variables. It is similar to the case “simulate the random inputs to the CS”. The random simulation express exactly the fact that the values are nondeterministically decided by the environment.

(g) Simplify reasonably the computation of the SRS. (Similar to the case of the CS.)

(h) Use embedded C code if necessary. (Similar to the case of the CS.)

(3) The second task is to define the position for the communication between the CS and the SRS. Generally, the usual location is between the assignment of key variables monitored by the SRS and the manipulation of physical equipment. Therefore, the SRS can check whether the system will be safe if the next manipulation is executed. If no, the SRS can send a message to activate the error processing functions.

(4) The third task is to define the function of error processing. This step is different for different projects, because it is based on the requirements of a specific system. In fact, the correctness of error processing also plays an important role in the correctness of the overall system and the functional validity of the SRS.

(5) Some Experiences.

(a) Check the exactitude of the established model, by using simulation mode. (Similar to the case of the CS.)

(b) Check the exactitude of the correctness properties, by using verification mode. We must make sure that the overall system (including the CS and the SRS) is safe, i.e., satisfies the specified correctness properties. If we aim at creating or modifying an SRS for identified risks, we must make sure that the overall system can avoid the previously detected errors, since the SRS component and additional error processing functions have been added.

If errors are detected, we must check the design of the SRS, and also the exactitude of the correctness properties (because they may specify a semantics different from what we expect).

Implement the SRS. We implement the SRS using the established model and computation. Note that faults may also occur at this step. Since numerous guidelines exist in industrial sectors to handle this issue, the discussion on reliable implementation is beyond the scope of this paper.

IV. Related Work

The use of formal methods and model checking for ensuring safety or proving the absence of certain hazards is not new in the literature.

For example, Eriksson [13] showed how formal verification can be used in a retrospective safety case through an example taken from railway signalling. This approach can be used to demonstrate the safety of the safety-critical systems in operation which were developed according to old practises that would be regarded as unacceptable today. In this application of formal methods, several particular problems were discussed, such as uncertainty about the original requirements and the required safety level of the various system functions.

The ForMoSA project [14] developed an integrated approach for safety analysis of critical embedded systems. The approach brings together the best of engineering practice, formal methods and mathematics: traditional safety analysis, temporal logics and verification, and statistics and optimization. These three orthogonal techniques cover three different aspects of safety: fault tolerance, functional correctness and quantitative analysis.

However, these works in the literature only focus on how to ensuring safety, rather than the functional validity of SRS in IEC 61508. Note that one key idea of the standard is the separation of the CS and the SRS. Thus ensuring the functional validity of SRS is different from the safety issues of a single control system which are discussed in the literature. Therefore, we are the first to consider the technical solution to functional validity of SRS in the literature.

Furthermore, our related papers [15][16] proposed the theory of formal control systems, based on the traditional automata theory. A formal control system consists of two automata. The controlled automaton is monitored by the controlling automaton to satisfy the given specification. The theory can be considered as the theoretical foundation of the SRS. The interested reader is referred to [15][16], since the theoretical aspect is beyond the scope of this industrial practice-oriented paper.

V. Conclusion

This paper first proposed the concept of *functional validity* of SRS, based on our observation on two important problems that occur in industrial practice, i.e., the rightness of overall and allocated safety requirements and the lack of technical methodologies for validating SRS.

Then this paper proposed a generic technical methodology (or a framework) which is based on computer-aided design

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-149-6

in association with automated verification tools, for designing *functionally valid* SRS. To the best of our knowledge, we are the first to consider the technical solution to functional validity of SRS in the literature.

The case study provided a customized demonstration of applying the methodology. The same methodology may be used for other situations by taking into account only some necessary adaptations, since it is a generic modeling approach for designing functionally valid SRS.

There are also some limitations of the proposed approach. First, the approach cannot completely eliminate design defects. This is due to the fact that it is not possible to identify all the safety requirements and correctness properties before verification. As a solution, we may use some sophisticated methodologies discussed in the literature to analyze hazards and their probabilities, e.g. fault tree analysis. Second, the modeling process is generally manual. We may consider how to automate the modeling process to decrease the time required for validation. Third, like other model checking techniques, the state explosion problem is also a challenge for scaling up the approach. One possible solution is to improve the model checking algorithm to reduce the state space [17]. Another possible solution is to develop optimized and customized model checker for verifying SRS.

Acknowledgment

This work was supported by the China Postdoctoral Science Foundation and the National Natural Science Foundation of China (60703033 and 61033002). The authors are grateful to the anonymous referees for their detailed comments and helpful suggestions.

References

[1] IEC, *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. International Electrotechnical Commission, 1999.

[2] F. Redmill, “IEC 61508 - principles and use in the management of safety,” *Computing & Control Engineering Journal*, vol. 9, no. 5, pp. 205–213, 1998.

[3] S. Brown, “Overview of IEC 61508 - design of electrical/-electronic/programmable electronic safety-related systems,” *Computing & Control Engineering Journal*, vol. 11, no. 1, pp. 6–12, 2000.

[4] R. Faller, “Project experience with IEC 61508 and its consequences,” *Safety Science*, vol. 42, no. 5, pp. 405–422, 2004.

[5] D. S. Herrmann, *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*. IEEE Computer Society, 2000.

[6] H. Gall, “Functional safety IEC 61508 / IEC 61511 the impact to certification and the user,” in *Proceedings of the 6th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2008)*. IEEE, 2008, pp. 1027–1031.

[7] D. Fowler and P. Bennett, “IEC 61508 - a suitable bases for the certification of safety-critical transport-infrastructure systems??” in *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2000)*, ser. Lecture Notes in Computer Science, F. Koornneef and M. van der Meulen, Eds., vol. 1943. Springer, 2000, pp. 250–263.

[8] T. Kletz, “Human problems with computer control,” *Plant/-Operations Progress*, vol. 1, no. 4, 1982.

[9] N. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.

[10] G. J. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[11] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000.

[13] L. henrik Eriksson, “Using formal methods in a retrospective safety case,” in *Proceedings of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2004)*, ser. Lecture Notes in Computer Science, vol. 3219. Springer, 2004, pp. 31–44.

[14] F. Ortmeier, A. Thums, G. Schellhorn, and W. Reif, “Combining formal methods and safety analysis - the formosa approach,” in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science, vol. 3147. Springer, 2004, pp. 474–493.

[15] Z. Chen and G. Motet, “Towards better support for the evolution of safety requirements via the model monitoring approach,” in *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010)*. ACM, 2010, pp. 219–222.

[16] Z. Chen and G. Motet, “System safety requirements as control structures,” in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*. IEEE Computer Society, 2009, pp. 324–331.

[17] Z. Chen and G. Motet, “Nevertrace claims for model checking,” in *Proceedings of the 17th International SPIN Workshop on Model Checking of Software (SPIN 2010)*, ser. Lecture Notes in Computer Science, vol. 6349. Springer, 2010, pp. 162–179.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-149-6