Automated Addition of Fault-Tolerance to SCR Toolset: A case study[1]

Fuad Abu-Jarad Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
3115 Engineering Building, East Lansing, MI 48824, USA
Email: {abujarad,sandeep}@cse.msu.edu
Web: *http://www.cse.msu.edu/~{abujarad,sandeep}*

Abstract

Automated addition of fault-tolerance to existing programs is highly desirable, as it allows the designer to focus on the system behavior in the absence of faults and leave the fault-tolerance aspect to automated techniques that guarantee correctness by construction. Automated addition of fault-tolerance is expected to be more successful if it is done *under the hood*, i.e., where the designer can continue to utilize existing tools and the addition of fault-tolerance is orthogonal to the tools that they use. This will reduce the learning curve for adding fault-tolerance as well as make addition of fault-tolerance across different design tools. With this motivation, in this paper, we focus on automated addition of fault-tolerance to the SCR tools. We illustrate our approach using two case studies: an altitude switch controller and an automobile cruise controller.

Keywords: SCR tools, Automated addition of fault-tolerance, Event-driven systems

[1] Tel: +1-517-355-2387, Fax: +1-517-432-1061
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF equipment grant EIA-0130724, and a grant from Michigan State University.

1. Introduction

Fault-tolerance, a guarantee to provide a specified quality of service in the presence of faults, is one of the important requirements of safety-critical systems. In event-driven high assurance systems, safety properties are required to be satisfied in absence as will as in the presence of faults. This constraint makes the problem of automatic addition of fault-tolerance complex and hard to achieve. For example, in an aircraft Altitude Switch System if an altimeter fails, the system should tolerate the failure and generate the appropriate response to recover from faulty state. System requirements may change due to newly discovered faults or to new threats. Redesigning the system to account for those faults could be a very long and costly process. Hence tools are desirable to allow designers to add fault-tolerance to systems as automatically as possible. To be most effective, the automatic addition of fault-tolerance should be transparent to the designer and the learning curve for such tool should be small. Ideally the tool should allow the designer to add fault-tolerance using user-defined interface that is part of the current tool set that they are already familiar with. Based on this motivation, in this paper, we propose enhancing an existing tool for requirements specification and make it support the automatic addition of fault tolerance.

There are two ways to enhance existing tools to include the functionality of adding fault tolerance. The first approach can be done by rebuilding the design tool to include the automatic addition of fault tolerance. Whereas the second approach would be to integrate an existing design tool (SCR[1], RSML[2], STATEMATE[3]) with an existing fault-tolerance tool so that addition of fault-tolerance is independent of the design tool. In the former approach the usability of the enhancements is limited to that specific tool and no other tools can make use of it i.e., reusability for adding fault-tolerance in different tools is limited to non-existent. However, in the latter approach, techniques for adding fault-tolerance can be centralized in one tool and therefore reusable. Moreover, in the latter approach, it would be feasible to apply the work from one design tool to another. Also, the separation of concerns in the latter approach is beneficial for both tools. Since enhancement to both tools can be made independently updates and upgrades in one tool will not significantly affect the other. Finally, it is much cheaper to build an interface layer between both tools rather than rebuilding them again to combine them together. For these reasons in this work we have chosen the latter approach.

For the design tool, we use Software Cost Reduction (SCR) [4]. SCR is a set of formal methods for constructing and verifying requirements specification document. U.S. Naval Research Laboratory (NRL) developed SCR in the late 70s. Since then it has been used in constructing many critical mission systems. SCR was used to design and model A-7 aircraft and in documenting requirements of many other systems such as OFP for A-6 aircraft, the Bell telephone, submarines communication systems and nuclear plants. SCR specifies system requirements using tabular notion in a precise and compact way making it possible for the user to automatically model and analyze the requirements document to identify errors.

For the fault tolerance, we use FTSyn [5] for our work. FTSyn is a tool for adding fault-tolerance to programs that are fault intolerant. Since the problem of adding fault-tolerance to distributed programs are NP-complete, FTSyn uses heuristic-based approaches to automatically add fault-tolerance to programs. To interact with FTSyn the fault intolerant program's variables, invariants, specification and faults should be specified in text file with guarded command format. FTSyn then outputs the fault-tolerant program, which is also in guarded command format. Recently this tool has been extended to handle symbolic techniques [6], that can allow us to handle large state space. In particular in [6] authors have shown that state space of 2^{100} can be efficiently used in syntheses.

Contributions of the paper: The main contributions of the paper are as follows:

- We present a tool that combines SCR tools and FTSyn. In this tool, the output of the SCR tools is imported into the FTSyn in order to automatically add fault-tolerance specification. The output of FTSyn is then exported back to the SCR tools to obtain the fault-tolerant SCR specification.
- We illustrate our tool using two different event-driven applications the aircraft Altitude Switch controller and the automobile Cruse Control System. In both of those systems we used our tool to transfer requirements specification described in SCR to programs in FTSyn. Then after FTSyn had added the fault-tolerance to specifications the tool translated the specification back so that they can be visualized in SCR.

Organization of the paper: The rest of this paper is organized as follows: In section 2, we discuss the background of this work. It gives a brief description of the formal SCR methods; it also provides highlights of SCR tools. We also give a brief overview of FTSyn. In Section 3, we describe the approach used in transforming the SCR to guarded commands. Section 4 presents two experiments and examples of the transformation. We present related work in Section 5. Finally the conclusion and future work are described in Section 6.

2. Background

In this section, we describe event-driven systems by explaining how their requirements are described using SCR formal methods. Later, we give a brief description of the automatic addition of fault-tolerance and the FTSyn tool that we have used to automate the addition of fault tolerance.

2.1. SCR Formal Model

The SCR formal method describes requirements using tabular notation. Tables can describe systems in an easy to understand way that leaves no room for misinterpretation of specifications [7, 8]. The constructs of SCR are based on Paranas's [9] Four Variable Model. This model describes the desired functionality of an embedded system in terms of four variables. The *Monitored Variables* represent environmental quantities whose value can change the system behavior. The *Controlled Variables* represent environmental quantities whose value is changed by the system. The *Input,* represents the set of resources that the system uses to measure the value of monitored variables. The *Output*, represents the set of resources that the system uses to change the values of monitored quantities. Four relations are also used to relate system variables and represent constraints on those variables.

- ***NAT***: is the set of relations that describes the way system variables (*monitored* or *controlled*) values are restricted by the laws of the environment whether they are imposed by previously deployed systems or by the physical laws.
- ***REQ***: is the set of relations that defines the way in which the system will control the change of *controlled variables* quantities based on a change of *monitored variables* quantities.
- ***IN***: is the set of relations that maps the values of the monitored quantities to the values of the input variables.
- ***OUT***: is the relation that maps the value of the output variables to the value of the controlled quantities.

The IN and OUT relations describe the behavior of the input and output devices in some level of isolation that gives requirements specification the freedom to specify the observed system behavior without going into the details. Four more variables are also used in the constructs of the SCR. These are *modes*, *terms*, *conditions* and *events*. The *Mode Class* is a state machine whose states are called modes. Changes from one mode to another are triggered by events. The *Terms* are a representation of a group of input variables, mode classes or other terms in one single term. *Events* are triggered by a change in a system entity.

The system is represented as a state machine $\Sigma = (S, S_{\bullet}, E^m, T)$ where S is the state space; $S_{\bullet}$ is the starting state; E^m represent a change in the value of the monitored variables; T is the function that determines the state S', the after state, based on given monitored event in E^m and a given state in S [2].

The SCR tools are a set of tools for constructing and validating requirements specifications. It's composed of a specification editor, a user interface for creating and editing the specification in a tabular way, a dependency graph browser, which uses the directed graph representation to show the dependency of variables, Simulator, which uses a symbolic variable representation to test if the desired system behavior is satisfied. The SCR tools also include different kinds of checkers: consistency checker, model checker, and property checker. This set of tools help systems designers to check and analyze the specifications and to automatically detect errors and missed case.

There are two major advantages of the SCR tools first; all the tools can interface with each other automatically and can behave as single application. Second, the toolset has been adopted by the industry and was used in the development of many real world applications. The toolset stores the specifications in

an ASCII text file from which other systems can have access to the specifications. This file is used as an interface channel to communicate with FTSyn framework.

2.2. Automatic Addition of Fault-Tolerance

Programs are subject to faults that may not be preventable. A program may function correctly in the absence of faults. However, it may not give the desired functionality when faults occur. The automatic addition of fault-tolerance is the process of the automatic transformation of a fault-intolerant program to a fault-tolerant one. This transformation guarantees that the program continues to satisfy the desired specification in the presence of faults.

FTSyn is a framework for adding fault-tolerance to fault-intolerant programs[5]. In FTSyn programs are represented in guarded commands language. This representation is the input of the FTSyn synthesis framework. The same language is also used in representing faults. FTSyn takes both the program and the faults as an input and generates the fault-tolerant program version as an output. Both the input and the output of FTSyn are ASCII text files in guarded command language.

To add fault-tolerance, FTSyn first identifies states from where faults alone can violate safety specification. It removes such states and transitions that reach them. Then, it adds recovery transitions to ensure that after the occurrence of faults, the program recovers to its legitimate states that are specified by its invariant. FTSyn also enables synthesis of distributed programs by allowing modeling of read/write restrictions of variables and ensuring that these restrictions are met in the synthesized program.

3. Integration of SCR and FTSyn

The integration of SCR and FTSyn mainly focuses on the mode table since the mode table captures the system behavior in response to different inputs. Hence, mode table is the most relevant in terms of the effect of the faults on system behavior. The integration focuses on translating the mode table so that it can be used as an input in FTSyn and then translating the FTSyn output so as to generate the mode table of the fault-tolerant SCR specification.

We illustrate the mode table in SCR using a simple example of a mode table for SCR specification (cf. Table 1). As the name suggests, this table describes different modes of *mRoom* and how they change in response to the events. *mRroom* has two modes: *Dark* and *Light* and one monitored variable *mSwitchOn*. This system switches the room from *Dark* mode to *Light* mode if the event *@T(mSwitchOn)* occurs, i.e. if the monitored variable *mSwitchOn* changes its value from false to true.

The mRoom Mode Table		
Old Mode	**Event**	**New Mode**
Dark	@T(mSwitchOn)	Light
Light	@F(mSwitchOn)	Dark

Table 1. mRoom Mode Table

Translating SCR specification to obtain an input for FTSyn is straightforward for most of the SCR specification. For example, *modes* are translated to *states*, *conditions* are translated to *guards* and mode *transitions* are captured by the before (guard) and the after state (statement) of the guarded command.

One part that needs special consideration is the events. Events in SCR occur at the time when the value of their condition is switched from false to true or vice versa in a single transition. It is not only the current state of the monitored variable that initiates the transition rather it is the combination of both the current and the old states. The notation used to represent events is as follows:

$$\textbf{@T(c) when d} \equiv \neg\ c \wedge c' \wedge d$$

, where(c) represents the condition value in the before state and (c') represent the condition value in the after state [2]. For example, if we consider the SCR mode table entry in mRoom table:

From “Dark” EVENT “@T (mSwtichOn)” TO “Light”

In the “before” state, the mode value *mRoom* is *Dark* and the condition *mSwitchOn* is *False.* And, in the “after” state the mode value *mRoom = Light* and the condition *mSwitchOn = True.*

In FTSyn (guarded commands) transition are represented in the following format:

$$(g \rightarrow st)$$

The guard, g, is a predicate whose value must be true in the before state in order for the statement, st, to execute. The guarded command translation for mRoom table entry would be:

$$(mRoom == Dark) \& \&(mSwtichOn == False) \rightarrow mRoom = light; \quad mSwtich = True;$$

The scenario of the translation between the SCR and guarded commands is described in Figure 1. The cycle begins at 1 by creating the specifications requirement using the SCR tools. The specifications in SCR formats are exported from the SCR tool set as in step 2. In step 3, the middle-layer imports the SCR specifications and the first translation phase generates an output file for the use in the addition of fault-tolerance by FTSyn. This file is imported in step 4 to FTSyn. FTSyn generates a fault-tolerant version of the program in step 5. In step 6, the middle-layer imports the FTSyn output and translates it back to SCR specification. Finally, in step 8, the file is imported back into SCR tool set so that it can be visualized using the SCR tools.

Thus, the translation layer shown in Figure 1 allows the automated addition of fault-tolerance where the addition is done *under the hood*. Thus, it allows users of the SCR tools to add fault-tolerance to specifications without knowing the details of FTSyn or the theory on which FTSyn is based.

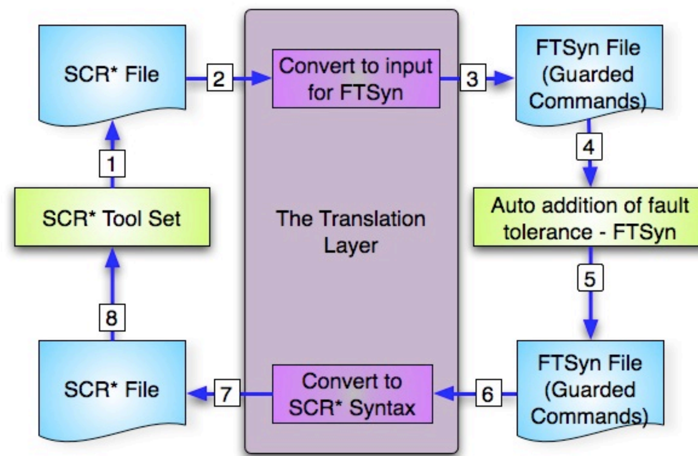

Figure 1. The cycle between SCR tools and FTSyn

4. Case Study

To illustrate the integration of SCR and FTSyn, we consider two systems: the control system for an aircraft Altitude Switch and the automobile Cruise Control System. For both systems, we briefly describe the concept and demonstrate the translation of the fault-intolerant SCR specification into an input for FTSyn and the translation of the fault-tolerant FTSyn output into the corresponding fault-tolerant SCR specification.

4.1 Altitude Switch

The Altitude Switch (ASW) system is responsible for turning on a Device of Interest (DOI) when the aircraft altitude is below 2000 feet. The finite state machine diagram of ASW is shown in Figure 2. It depends on five monitored variables mAltBelow, mDOIstatusis, mInitializing, mInhibit and mReset. Variable mAltBelow is a boolean variable and its value is true when the aircraft descends below a certain height (2000 feet). Variable mDOIstatus is true when the DOI is on. Variable mInitializing indicates if the system is being initialized. And, mInhibit indicates whether the system can turn on the DOI or not. The mReset monitors the reset. Now, we show how fault-tolerant altitude switch controller is synthesized using the tool described in Figure 1.

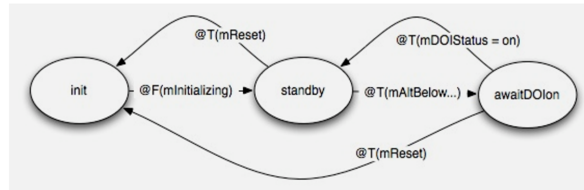

Figure 2. The ASW state machine diagram

Step 1: As shown in Figure 1 at step 1, we extract mode table of ASW system in SCR specification. The *mcStatus* mode table of the ASW system is illustrated in Figure 3. It describes mode class *mcStatus* that represents a function between the *monitored variables* and the current value of the mcStatus. The mcStatus class has one of the following three modes: standby, init, or awaitDOIon. For example, the first entry in the table shows that if the mInitializing becomes false and the mcStatus is equal to init then the new value of the mcStatus = standby.

Fault intolerant mode class mcStatus		
Old Mode	Event	New Mode
init	@F(mInitializing)	Standby
standby	@T(mReset)	init
standby	@T(mAltBelow)WHEN NOT mInhibit AND mDOIStatus=off	awaitDOIon
awaitDOIon	@T(mDOIStatus=on)	standby
awaitDOIon	@T(mReset)	init

Figure 3. Mode Transition Table for mcStatus

Step 2&3: At step 2, we import the SCR specification into the middle layer, which generates specification in FTSyn format at step 3. The result of the translation layer is in Figure 4. The first entry in Figure 4 shows that the old value of the mcStatus should be equal to standby, and mReset is False in the "before" state in order to execute the corresponding statement. The two statements in the right hand side represent the "after" state; both values of mcStatus and mReset should be changed.

```
((mcStatus==init)  &&  ((mInitializing) == True ) )  ->  mcStatus=standby;  (mInitializing) ==False ;
((mcStatus==standby)  &&  ((mReset) == False ) )  ->  mcStatus=init;  (mReset) =True ;
((mcStatus==standby)  &&  ((mAltBelow) == False  &&  !mInhibit && mDOIStatus=off) )  ->
                                        mcStatus=awaitDOIon;  (mAltBelow) = True  &&  !mInhibit && mDOIStatus=off;
((mcStatus==awaitDOIon)  &&  ((mDOIStatus=on) == False ) )  ->  mcStatus=standby;  (mDOIStatus=on) = True ;
((mcStatus==awaitDOIon)  &&  ((mReset) == False ) )  ->  mcStatus=init;  (mReset) = True ;
```

Figure 4. The mcStatus mode table translated

We consider three hardware malfunctions that may alter the operation of the fault intolerant ASW controller[4]. They are an altimeter fault, an initialization fault and DOI fault. All three faults are time-out faults, i.e., they require the system to stay in a given state for a specified amount of time. But since FTSyn dose not include the notion of time yet, we abstract those faults to be an *on/off* flags. We added a new mode, *fault,* to the mode class to indicate the presence of faults in the system. Figure 5 shows how those faults are represented in the input file to FTSyn.

```
(mcStatus = init ) && (Init_Duration_Fualt == True )      -> Init_Duration_Fualt = False; mcStatus = Fault;
|
(standby = init ) && (Alt_Duration_Fault == True )        -> Alt_Duration_Fault = False; mcStatus = Fault;
|
(awaitDOIon = init ) && ( AwaitDOI_Duartion_Fault == True ) -> AwaitDOI_Duartion_Fault = False; mcStatus = Fault;
```

Figure 5. The FTSyn Fault section

Step 4: At step 4, we use the translated SCR specification and the three faults described in Figure 5 as an input to FTSyn so that FTSyn can add fault-tolerance to ASW specification that will tolerate the failure of the altimeter, initialization or DOI.

Step 5: The result of step 5 is shown in Figure 6. The parts were FTSyn have add the tolerance were at two places. First, the condition (mAltFail == False) was added to the third transition guard to prevent the mcStatus from transitioning to faulty state. Second, the last transition was added to lead to recovery from the fault state to one of the system safe states.

```
(mcStatus==init)              &&  ((mInitializing) == True ) )     ->  mcStatus=standby;  (mInitializing) = False ;
((mcStatus==standby)          &&  ((mReset) == False ) )           ->  mcStatus=init;  (mReset) == True ;
((mcStatus==standby)          &&  ((mAltBelow) == False  &&  !mInhibit && mDOIStatus=off && mAltFail == False) )
                                                                   ->  mcStatus=awaitDOIon;  (mAltBelow) == True  ;
((mcStatus==awaitDOIon)  &&  ((mDOIStatus == False ) )             ->  mcStatus=standby;  (mDOIStatus  = True );
((mcStatus==awaitDOIon)  &&  ((mReset) == False ) )                ->  mcStatus=init;  (mReset) == True;
(( mcStatus == fault)         &&  ( ( mReset) == False  )   )      ->  mcStatus = standby; mReset = True ;
```

Figure 6. The fault-tolerant mcStatus mode table

Step 6 & 7: We import the FTSyn specifications into the translation layer at step 6 to translate it to a fault-tolerant SCR specifications. Figure 7 is the result after applying the translation on the mcStatus from FTSyn output to SCR.

Step 8: In step 8, we import back into SCR tools the fault-tolerant SCR specifications. The fault-tolerant specifications are as shown in Figure 7.

Fault-tolerant mode class mcStatus		
Old Mode	Event	New Mode
init	@F(mInitializing)	Standby
standby	@T(mReset)	init
standby	@T(mAltBelow)WHEN NOT mInhibit AND mDOIStatus=off AND NOT **mAltFail**	awaitDOIon
awaitDOIon	@T(mDOIStatus=on)	standby
awaitDOIon	@T(mReset)	init
fault	**@T(mReset)**	**init**
init	**Init_Duration_Fualt**	**fault**
standby	**Alt_Duration_Fault =**	**fault**
awaitDOIon	**AwaitDOI_Duartion_Fault**	**fault**

Figure 7. Fault-tolerant mode class mcStatus

We note that in [4] the addition of fault-tolerance to the SCR specifications was done manually. They have manually added both the faults and the recovery to the SCR mode table. They wanted to show that mode table are capable of reporting and handling hardware malfunctions. The addition of fault-tolerance in this approach depends on the domain knowledge and how much the analyst knows about the problem.

4.2 Cruise Control

The cruise control system (CCS)controls the cruising speed of the automobile by controlling the throttle position. It depends on several monitored variables like *mIgnon, mEngRunning, mSpeed, mLever and mBreake*. The system uses monitored variables values to control the automobile speed. It can be engaged by setting the *const* switch to “on”, provided that other conditions like engine running and ignition is on are met. The CCS can maintain constant, decrease, or increase automobile speed depending on the current speed. Now, we show how fault-tolerant CCS is synthesized using the tool described in Figure 1.

Step 1: The mCruise mode table is shown in Figure 8. The table specifies the values that the mCruise class can take. The second entry in this table indicates that if mCruise was Inactive and the engine was turned off, then the mCruise value should be set to “off”.

Fault intolerant mode class mcCruise		
Old Mode	Event	New Mode
Off	@T(mIgnOn)	Inactive
Inactive	@F(mIgnOn)	Off
Inactive	@T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise
Cruise	@F(mIgnOn)	Off
Cruise	@F(mEngRunning)	Inactive
Cruise	@T(mBrake) OR @T(mLever = off)	Override
Override	@F(mIgnOn)	Off
Override	@F(mEngRunning)	Inactive
Override	@T(mLever = resume) WHEN mIgnOn AND mEngRunning AND NOT mBrake OR @T(mLever = const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise

Figure 8. Fault intolerant mode class mcCruise.

Step 2&3: At step 2, we imported the mode table in Figure 8 into the middle layer, which generated specification in FTSyn format at step 3. Figure 9 is the result of translating the mCruise mode table to FTSyn. It can be seen that after the implication sign, there are two statements; one is the new value of the mode class and the other is the "after" state value of the predicate.

```
((mcCruise==Off)        &&  (NOT(mIgnOn)) )  ->  mcCruise=Inactive AND Not( NOT(mIgnOn));
((mcCruise==Inactive)  &&  ((mIgnOn)) )    ->  mcCruise=Off AND Not( (mIgnOn));
((mcCruise==Inactive)  &&  (NOT(mLever=const) AND mIgnOn AND mEngRunning AND NOTmBrake) )
            ->  mcCruise=Cruise AND Not( NOT(mLever=const) AND mIgnOn AND mEngRunning AND NOTmBrake);
((mcCruise==Cruise)    &&  ((mIgnOn)) )    ->  mcCruise=Off AND Not( (mIgnOn));
((mcCruise==Cruise)    &&  ((mEngRunning)) )  ->  mcCruise=Inactive;Not( (mEngRunning));
((mcCruise==Cruise)    &&  (NOT(mBrake)ORNOT(mLever=off)) )
            =>mcCruise=OverrideANDNot(OT(mBrake)OR NOT(mLever=off));
((mcCruise==Override) &&  ((mIgnOn)) )     ->  mcCruise=Off AND Not( (mIgnOn));
((mcCruise==Override) &&((mEngRunning)) ) ->  mcCruise=Inactive AND Not( (mEngRunning));
((mcCruise==Override)  &&  (NOT(mLever=resume) AND mIgnOn AND mEngRunning AND NOTmBrakeORNOT(mLever=const)
AND mIgnOn AND mEngRunning AND NOTmBrake) )
            =>mcCruise=Cruise Not( NOT(mLever=resume) AND mIgnOn AND mEngRunning AND
            NOTmBrakeORNOT(mLever=const) AND mIgnOn AND mEngRunning AND NOTmBrake);
```

Figure 9. The translated Pressure mode table

We consider a system malfunctions that may alter the operation of the fault intolerant CCS. The fault takes place when the status of the cruise becomes unknown. Figure 10 shows how this fault is represented in the input file to FTSyn.

```
(mcCruise == Override) || (mcCruise == Cruise) || (mcCruise == Inactive) || (mcCruise == Off)  && ( CruiseFault == True) ->
mcCruise = Unkown; CruiseFault = False;
```

Figure 10. The FTSyn Fault section

Step 4:AT step 4, we have inputted the faults and the fault intolerant CCS to FTSyn in order to add fault-tolerance to the CCS system to tolerate a recover from "unknown" state to one of the CCS safe state.

Step 5: The result of step 5 is shown in Figure11. We can see how the fault is being considered and the recovery from the fault state is taking the system to a safe state. FTSyn added two actions to recover from the unknown state to one of the system valid states depending on the value of the IgnOn monitored variable.

```
( ( mcCruise == Off )      &&  ( ! ( mIgnOn )  )  )-> mcCruise = Inactive ;
( ( mcCruise == Inactive )  &&  ( ( mIgnOn ) )  )          ->  mcCruise = Off ;
( ( mcCruise == Inactive )  &&  (  ! ( mLever == const )    &&  mIgnOn && mEngRunning &&  ! mBrake ) )
                 ->  mcCruise = Cruise ;
( ( mcCruise == Cruise )    &&  ( ( mIgnOn )  ) ) ->  mcCruise = Off ;
( ( mcCruise == Cruise )    &&  ( ( mEngRunning )   ) ) ->  mcCruise = Inactive ;
( ( mcCruise == Cruise )    &&  ( ! ( mBrake )   ) || ( ! ( mLever == off )  ) )  ->  mcCruise = Override ;
( ( mcCruise == Override )  &&  ( ( mIgnOn )   ) ) ->  mcCruise = Off ;
( ( mcCruise == Override )  &&  ( ( mEngRunning )  ) )  ->  mcCruise = Inactive ;
( ( mcCruise == Override )  &&  ( ! ( mLever == resume )   &&  mIgnOn && mEngRunning &&  ! mBrake  ) && ( ! ( mLever ==
const )  && mIgnOn  && mEngRunning &&  ! mBrake ) )  ->  mcCruise = Cruise ;
( ( mcCruise == Unknown )   &&  ( !( IgnOn  ) ) ) ->mcCruise = Off ;
( ( mcCruise == Unknown )   &&  ( ( IgnOn  ) ) ) ->mcCruise = Inactive ;
```

Figure 11. The fault-tolerant cruise control program

Step 6 & 7: We have imported the FTSyn specification into the translation layer at step 6 to translate it to a fault-tolerant SCR specification. Figure 12 is the result after applying the translation on the mcStatus from FTSyn program to SCR tables.

Step 8: In step 8, we import back into SCR tools the fault-tolerant SCR specification.
The fault-tolerant specification is as shown in Figure 12.

Fault-tolerant mode class mcCruise		
Old Mode	Event	New Mode
Off	@T(mIgnOn)	Inactive
Inactive	@F(mIgnOn)	Off
Inactive	@T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise
Cruise	@F(mIgnOn)	Off
Cruise	@F(mEngRunning)	Inactive
Cruise	@T(mBrake) OR @T(mLever = off)	Override
Override	@F(mIgnOn)	Off
Override	@F(mEngRunning)	Inactive
Override	@T(mLever = resume) WHEN mIgnOn AND mEngRunning AND NOT mBrake OR @T(mLever = const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise
Unknown	**@T (IgnOn)**	**Off**
Unknown	**@F (IgnOn)**	**Inactive**
Override, Cruise, Inactive, Off	**@T(CruiseFault)**	**Unknown**

Figure 12 Fault-tolerant mode class mcCruise

6. Conclusion and Future Work

In this paper, we outlined the architecture and the design of automatic addition of fault-tolerance to the requirements specification of event-driven systems described with SCR formal methods. We achieved this by integrating FTSyn with the SCR tools. FTSyn will collaborate with SCR tools to automatically add fault-tolerance to programs at design time. We have developed and tested a middle-layer that can convert the SCR specification to FTSyn specification. Systems designers who use SCR tools will have the functionality, under-the-hood, for adding fault-tolerance to their requirements specifications. At the same time we will give the FTSyn user the ability to use SCR tools to formalize their programs in a user-friendly application before they attend to add fault-tolerance to them.

This work allows the designer to perform automated addition of fault-tolerance without significant knowledge about FTSyn or how it adds fault-tolerance in an automated fashion. In particular, FTSyn expects four inputs, the fault-intolerant program, faults, specification and the invariant. Of these, fault-intolerant program and faults can be specified using SCR tables and imported automatically. We are currently developing heuristics that would allow the invariant to be generated based on the initial states provided by SCR specification. Thus, the designer would only have to focus on specifying requirements that need to be met in the presence of faults. In this context, we would like to note that automated synthesis with FTSyn provides the possibility of detecting errors in requirements themselves. In particular, in our work on altitude switch controller, we considered the case where no constraints are specified on how recovery could be added. This caused several recovery paths to be added, e.g., to states such as awaitDOIon. However, this conflicted with the requirement that the recovery can only be to the Init state. Since FTSyn tries to provide maximum non-determinism in the synthesized program, it provides the potential when requirements are missing.

Future work in this context will focus on adopting such transformation to the complete SCR specifications and not only to the modes. We will also research the possibility of integrating FTSyn to other requirements specification tools like RSML and STATEMATE. We also intend to study additional examples on automatic addition of fault-tolerance for requirements documents built using SCR tools.

Reference

1. Heitmeyer, C., Labaw, B., and Kiskis, D., *Consistency checking of SCR-style requirements specifications.* Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on, 1995: p. 56-63.
2. Heimdahl, M.P.E. and Leveson, N.G., *Completeness and consistency in hierarchical state-based requirements.* IEEE Transactions on Software Engineering, 1996. **22**(6): p. 363-377.

3. Harel, D. and Naamad, A., *The STATEMATE semantics of statecharts.* ACM Transactions on Software Engineering and Methodology (TOSEM), 1996. **5**(4): p. 293-333.
4. Bharadwaj, R. and Heitmeyer, C., *Developing high assurance avionics systems with the SCRrequirements method.* Digital Avionics Systems Conferences, 2000. Proceedings. DASC. The 19th, 2000. **1**.
5. Ebnenasir, A. and Kulkarni, S.S., *A framework for automatic synthesis of fault-tolerance.* International Journal of Software Tools for Technology Transfer, 2005.
6. Bonakdarpour, B. and Kulkarni, S.S., *Exploiting Symbolic Techniques in Automated Synthesis of Distributed Programs with Large State Space* International Conference on Distributed Computing Systems, Toronto Canada, 2007.
7. Heitmeyer, C. and Jeffords, R., *Applying a Formal Requirements Method to Three NASA Systems: Lessons Learned,.* Proceedings of the 2007 IEEE Aerospace Conference, Big Sky, MT, 2007.
8. Heitmeyer, C., Kirby, J., and Labaw, B., *Tools for formal specification, verification, and validation of requirements.* Proc. 12th Annual Conf. on Computer Assurance (COMPASS'97), Gaithersburg, MD, June, 1997.
9. Parnas, D.L. and Madey, J., *Functional documents for computer systems.* Science of Computer Programming, 1995. **25**(1): p. 41-61.