A Specification-Driven Approach to Embedded FDIR Code Generation*

Federico Bonafini[1], Roberto Cavada[2], Alessandro Cimatti[2], Guillermo Gomez[2], and Stefano Tonetta[2]

[1] Innova Engineering, Tione, Italy
[2] Fondazione Bruno Kessler, Trento, Italy

Abstract. Fault Detection, Isolation, and Recovery (FDIR) components are essential for managing faults and ensuring safety and reliability in safety-critical applications. This paper presents a specification-driven approach to the automatic generation of embedded FDIR code. Our method leverages formal specifications of fault conditions and recovery procedures to synthesize fault detection and recovery mechanisms, reducing manual coding and the potential for human error. The proposed toolchain translates high-level specifications into platform-specific embedded code, while model checking can be used to validate and verify the FDIR logic. We detail the underlying architecture, the specification language, and the code generation process, highlighting the flexibility and scalability of the approach. Through a case study in the energy domain, we demonstrate the tool's ability to handle complex fault scenarios, improve development efficiency, and enhance system reliability.

1 Introduction

Embedded software-based systems have enabled the implementation of complex control functionalities, ensuring energy-efficient and adaptive operations, and a high degree of parameterization. Among these capabilities, *fault management* is particularly crucial, especially in safety-critical systems, as it ensures continued operation even in degraded, non-nominal conditions. At design time, hazard analysis and safety assessments are conducted, with fault prevention and recovery strategies implemented through monitoring and reconfiguration procedures, as well as redundant equipment. These safety mechanisms are often then handled by control software, which provides high degree of parametrization and reuse.

The concept of FDIR encompasses a set of functions for fault management that include detecting the occurrence of a fault, identifying it, and applying the appropriate recovery actions. FDIR interacts with the plant to read input data and with the controller to command reconfiguration of redundant components or other recovery procedures. FDIR can be conceptually divided into two main

* The work is financed by the Autonomous Province of Trento in scope of L.P. No. 6/1999 with determination. No. 592 of 09/08/2021. – Ref.: 2021-AG12-00783. - project NPDCR (New residential heat pump)

components (see Fig. **??**): Fault Detection and Isolation (FDI), which focuses on detecting and identifying faults, and Fault Recovery (FR), which applies the most suitable response based on the detected fault.

Designing FDIR modules presents significant challenges. On one hand, FDI often requires to monitor complex temporal extended conditions. On the other, defining appropriate recovery actions requires covering all possible fault combinations and arranging them based on their priorities. In fact, faults can occur even during the execution of a recovery procedure intended to address a different issue with lower criticality. While model-based safety analysis techniques have been proposed to design FDIR components with formal methods, they neglect the complexity of software interfaces and the need of the FDIR software to interact with procedures and data structures provided by the platform.

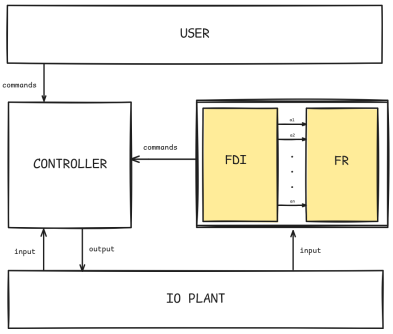

Fig. 1. FDIR function overview

In this paper, we propose a novel end-to-end approach to the development of FDIR software. The idea is to start from a high-level specification and automatically generate platform-specific code suitable for deployment on embedded controllers with real-time constraints, thereby reducing manual effort and minimizing human error. The advantages of the approach are manifold: first, it can handle many fault conditions avoiding entangled solutions, by separating monitoring from recovery, and defining a clear priority on the concurrent recovery actions; second, it can be adapted to different platforms; third, it allows for the application of formal verification techniques to check the FDIR logic.

The approach has been devised in the context of a long-term collaboration between Innova Engineering (IE) and FBK, and was driven by the goal of defining a structured software development process customized for the portfolio of IE products. We discuss its application to a real-world case study, a heat pump controller that is representative of the most recent products of the company.

The key contributions of this work are as follows:

- A specification-driven methodology for defining fault conditions, detection mechanisms, and recovery procedures in a structured and formalized manner.
- An automated toolchain that translates high-level specifications into platform-specific embedded code.
- A principled integration of formal verification through model checking techniques, enabling rigorous validation of FDIR logic before deployment.
- A case study in the energy domain, demonstrating the effectiveness of our approach in handling complex fault scenarios while enhancing reliability.

2 Related works

A wide literature studied model-based fault detection and diagnosis, which leverage mathematical models of system dynamics. There are two primary approaches: one rooted in control theory and the other based on logic and artificial intelligence (AI) (see [**?**]). The control-theoretic approach [**?**,**?**,**?**] [**?**,**?**,**?**,**?**] relies on dynamic system models (e.g., system of differential equations), and use methods such as state observers, Kalman filters, and parity relations to detect and isolate faults.

The logic and AI-based approach, in contrast, leverages formal methods, expert systems, and machine learning techniques for fault detection and diagnosis. Model-based diagnosis [**?**,**?**] approaches use logic satisfiability to check consistency between expected and observed behavior. Model-Based Safety Analysis (MBSA) uses model checking techniques to analyze the design of safety mechanisms [**?**], including FDIR components. It starts from a component-based model with high level view of the component behaviors, and inject faults in these models to validate the reaction of the FDIR. This process has been integrated in standard aerospace development process [**?**] and in a ESA toolchain [**?**]. Recent work has also applied model checking to assess and improve fault tolerance of satellite systems [**?**], focusing on verifying existing logic at system level rather than synthesizing embedded-level components from formal specifications. Logic based specification of FDIR has been formalized in [**?**,**?**], dealing with the inherent epistemic problem of diagnosability. Finally, the monitoring of properties specified in temporal logic can be solved with runtime verification techniques [**?**].

In the above mentioned works, recovery mechanisms are also divided between control-theoretic strategies, such as adaptive and fault-tolerant control, and logic or AI-based strategies, such as planning or reinforcement learning. However, in both cases, most of the academic works are not adopted at industrial level [**?**]. One of the reasons is that they neglect the complexity of managing software procedures for recovery and handling the logic of switching between them, which is the main contribution of this paper. In general, there is a lack of guidance and tool support for implementing complex FDIR components at software level. An example of such efforts is the development of an FDIR software fault tree library for onboard computers [**?**], which focuses on reuse and structuring of fault handling logic, though without formal specification or automatic synthesis. In the direction of closing this gap, this paper proposes a specification-based methodology and a tool support applied to a real-world scenario.

3 Preliminary notions

3.1 Temporal Logics and Formal Verification

We adopt temporal logic [**?**] as a formalism for the specification of properties of execution traces. Temporal logics are common in formal verification [**?**,**?**] to represent different kinds of requirements. Properties over the states (e.g.,

"`temperature>threshold`") constitute the basic elements of formulas. Temporal operators such as "always in the future" (`G`) and "sometimes in the future" (`F`) can be nested to express complex temporal properties. For example, it is possible express FDIR properties, like requiring that an alarm is triggered whenever the temperature is too high ("`G ((temperature>threshold) implies alarm-high-temp)`"), or that a suitable response is eventually delivered ("`G ((temperature>threshold) implies F recovery-high-temp)`"). In addition to future time operators, it is often convenient to adopt past operators, such as "historically" (`H`) or "once" (`O`) in the past and that allow to express properties over past states. Finally, metric operators like in the last n steps (`H [0,n]`) are used to constrain an interval of time. In the following, we use LTL (with all operators mentioned above) for model checking, while runtime verification is applied to PastLTL, the fragment that uses only the past operators.

Model checking is a technique to verify whether the model of a system satisfies a property, i.e. all its traces satisfy the property. The model is typically described as a symbolic transition system and symbolic verification of temporal properties is performed with algorithms combining deduction and reachability through logic-based operations (see [**?**] for a survey).

Runtime verification [**?**] is a lightweight verification technique intended to analyze the observable signals of a running system. The idea is to evaluate a temporal logic specification of the property on the actual execution of the system. Then, the automata-theoretic approach produces an observer that is put in synchronous product to the system under observation and raises a flag then the current trace is violated. Runtime verification is considered to be appealing since it is logically well founded, does not alter the nominal behavior of the system under analysis, and supports various implementation patterns.

3.2 SDL, ASN.1 and OpenGeode

For code generation, we rely internally on SDL and ASN.1, and the OpenGeode tool support. SDL (Specification and Description Language) is an established standard formal language to describe both networks of communicating state machines at system level, and the specification of the behavior of Finite State Machine (FSM). SDL comes with both a graphical and textual notation for the specification of the system, the communication ports and connections, functional blocks definition, and behavior specification in FSM. Transitions are triggered by events such as reception of a queued message, or certain observed condition becoming true, or a timer expiring. Operations admitted in transitions include setting variables and timers, and sending events to other blocks through ports and connections among them. Conditions and operation can be organized in functions and procedures. State machines can be hierarchical, meaning that a given high level state of the machine can contain an inner state machine.

SDL is supported by various commercial and open source editors and compilers. OpenGeode is an open source graphical editor which features also automatic code generation in Ada and C. OpenGeode presents both some limitations and extensions to the SDL language. In particular it removes the sup-

port for the data type system of SDL, and provides instead support for the ASN.1 specification. ASN.1 is a formal specification used to describe data types with constraints, and constant values. The ASN.1 specification is compiled by ASN1SCC, an open source compiler for ASN.1 for data structures and encoding/decoding code, with multiple target languages like Ada, C and Scala.

The main advantage of using SDL + ASN.1 and OPENGEODE + ASN1SCC over other widespread languages and code generators (e.g. the common pair Stateflow/Matlab) is the well-described and clear formal semantics of SDL which greatly simplified the automatic translation of the FDIR specifications into a FSM based representation in SDL. Using SDL and ASN.1 as standard intermediate languages allowed us to rely on OPENGEODE and ASN1SCC to generate automatically the code, which proved to be easily integrated into both the host and target embedded platforms. However, SDL was not considered for the frontend specification language. Instead, we preferred a tabular format for the specification of the fault modes, in a form very similar to the FMEA tables (Failure Mode and Effects Analysis) which the domain experts are familiar with. For the same reasons, for the specification of the recovery procedures we preferred a syntactically limited language to describe the corresponding procedural steps.

4 Specification-Driven FDIR Code Generation

In this section, we describe the methodology that we propose to generate the FDIR code from a high-level specification. We start from giving an overview of the approach (Sec. **??**). We then detail the specification in terms of failure conditions (Sec. **??**), recovery procedures (Sec. **??**), and failure mode management (Sec. **??**). In Sec. **??** and **??**, we detail the process of generating respectively the code and the model for formal verification.

4.1 Overview of the Specification-Based Methodology

In this section, we describe the methodology that we propose to generate the FDIR code from a high-level specification and the the tool called FDIRGEN that supports it. The tools is publicly available at https://fm.fbk.eu/tools/fdirgen/. We start from giving an overview of workflow of the approach, depicted in Fig. **??**. The inputs are the artifacts provided by the user, i.e. the *controller interface* and the *FDIR specification*.

The controller interface consists of a set of declarations to define the signals and the primitives that will be available for the generated FDIR code. These include user-defined data types (e.g., structs, ranged integers), input and output variables, functions to perform calculations over the inputs, and functions representing commands writing on the outputs.

The FDIR specification is given by a table, defining monitoring conditions, recovery procedures and a Finite-State Machine (FSM) defining the operational modes of the FDIR. The specification is therefore hierarchical: the FSM defines

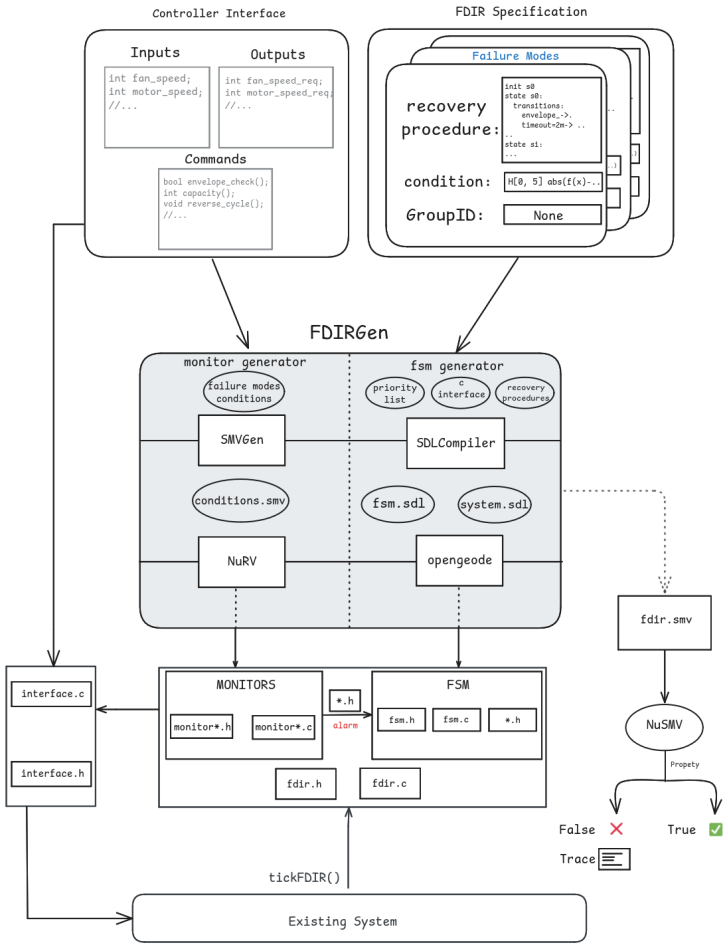

Fig. 2. Workflow of the Specification-Based Approach

the top-level modes and the transitions to switch between modes; each mode is then associated to a monitoring condition and to a recovery procedure. As shown in Fig. **??**, the FDIR specification table is composed of three main parts:

fault detection and isolation conditions defined in PastLTL and specified over input variables and input functions; (detailed in Sec. **??**)

recovery procedures defined as state machines extended with timeout constraints specified in a domain specific language; the procedures call the output functions to write on output variables; (Sec. **??**)

priorities and threshold groups determining the mode transitions. (Sec. **??**)

FDIR mode	Fault Detection Condition	Fault Isolation Condition	Priority	Threshold Group	Fault Recovery Procedure
Mode 1	Condition A	Condition X	High	Group 1	Restart System
Mode 2		Condition Y	Medium	Group 2	Switch to Backup
Mode 3		Condition Z	Low	Group 3	Alert Operator

Fig. 3. FDIR Specification Table structure.

The specification is given to the FDIRGen software, which generates the *FDIR code* that interacts with the system controller. The resulting FDIR code is composed by a *monitor module* and an *FSM module*. (Detailed in Sec. **??**)

The FDIRGen software can also generate a model specified in SMV, the input language of the nuXmv model checker. The model represents the same FDIR logic that is encoded in the generated code. Thus, the SMV model can be used to validate and verify the FDIR logic with model checking techniques. (Detailed in Sec. **??**)

Running Example We illustrate the approach through a running example. This is based on a simple specification, with two sources of power, the main one and the secondary that is activated in case of main failure. We want an FDIR system to be able to detect cases where we have been relying on the secondary pump for too long, and cases where both of the power supplies are failing.

Starting from the specification table, for the running example, we can define the following table to express the mentioned properties (we removed group id and isolation condition for space constraints):

FDIR mode	Fault Detection Condition	Priority	Fault Recovery Procedure

Mode 1	$H[0s, 10s] \wedge \neg\text{main}$ $\wedge\ H[0s, 20s] \wedge \neg\text{secondary}$	1	`init restarted [ restart_power(); ]`
			`state restarted:`
			`transitions:`
			`timeout = 75s -> fail`
			`main -> ok`

Mode 2	$secondary\ S[10s, 10s]!main$	2	Switch to Backup

Fig. 4. Running example FDIR specification

As far as interface is concerned, we need both attributes used on the properties and the recovery procedures (these procedures are the ones on the existing system, we just have to specify the header).

Input attributes:

```
bool main;
bool secondary;
```

Commands for the controller:

```
void recovery();
void restart_power();
```

4.2 Monitoring Conditions

The monitoring conditions are specified in an extended version of PastLTL over the input variables, where:

- in the atomic conditions, we can specify arithmetic constraints combining input variables with calls to the input functions (e.g., "`abs(expected_flow() -flow)>delta`", where `abs` is a macro for the absolute value;
- the time bounds of interval in the metric operators are expressed in time units that may be milliseconds (ms), seconds (s), or minutes (m).

The PastLTL operators allow to specify temporally extended conditions. Typical patterns of conditions that we use for monitoring are $H_I(\alpha_1)$ and $\alpha_1 \wedge O_I(\alpha_2)$ where α_1 and α_2 are atomic conditions. The first one identifies the situation in which a condition α_1 lasts continuously in the interval I. The second one identifies the situation in which we detect α_2 and in a previous moment, within the interval I, α_1 was true. Examples of these patters are:

```
H [0s, 30s] abs(expected_flow()-flow)>delta
```

which is true in the moment in which, for the last 30 seconds, the expected flow differed from the actual one more than a certain delta constant;

```
flow=0 & O [0ms, 20ms] flow>high_value
```

which is true in the moment in which the flow drops to zero from a high value that was read in the previous 20 ms.

In order to ease and organize better the specification of the conditions, they are divided into two columns of the specification table. The first column, called *fault detection condition*, should contain the conditions that identify the detection of a fault and trigger the alarm. The second column, called *fault isolation condition*, should contain the conditions that try to discriminate the cause of the problem and used to differentiate between one mode and another. Multiple isolation conditions can be associated to one fault detection condition. For example, to detect a fault we may check if the temperature is different from the set value and does not change. This may be caused by potentially different causes, which can be distinguished by checking if the fan is working and if there is power.

4.3 Recovery Procedures

The recovery procedures are specified in a domain-specific modeling language. This is a textual specification of states, transitions, and timeouts. The purpose is to provide a simple high-level specification of the procedures avoiding cumbersome code for input/output, timers or complex data structures. For example, in

order to update an output variable `x` increasing its value by another input variable `y`, we would simply write `x += y`; in SDL external variables cannot be used directly, so we would have to write a getter on each variable access, and a setter on every update; in our example, this would look like the following sequence: `get_x(x); get_y(y); set_x(x + y)`.

Thus, each recovery procedure provides a list of states, and for each state, a list of outgoing transitions that write on the output variables of the controller through the output functions specified in the controller interface and change the state of the procedure. An init statement represents the initial state of the procedure. Optionally, an effects section can follow, specifying initial conditions or setup actions. The procedure may define a set of local variables, followed by the declaration of states utilized within the procedure.

Each program has two implicit terminating states:

- ok: it indicates successful recovery and leads to a nominal system mode.
- fail: it denotes an unrecoverable error, leading to a stophold system mode.

The program terminates whenever one of these two states is entered.

For each state, the program lists a sequence of possible transitions. Each transition has a guard and an effect. The first transition with the guard that evaluates to true is taken, the effect is executed, which include to go to a new state or remain in the current one. Transitions between states are prioritized, with priority determined by the order in which transitions are declared. Each state may optionally specify an invariant condition that must hold upon entry. If the condition is violated, execution transitions immediately to the fail state.

States are equipped with local timeouts, which reset each time the state is re-entered and can be used in the guards of transitions. Additionally, timeouts can be defined over a set of states, resetting whenever execution enters this set from an external state. This feature is useful when modeling loops involving one or more states, allowing explicit control over their execution duration. As for the monitoring conditions, time constraints use time units.

The following listing gives a simple example of recovery procedure.

```
init check_velocity [ req_speed := 0; ]

timeout checking_loop : timeout=3m on
                        {check_velocity, check_temperature};

state check_velocity:
  invar: req_velocity > 0
  transitions:
    checking_loop -> fail
    timeout=2s -> check_velocity
    [velocity > req_velocity \ req_velocity += 2;]
                                    -> check_temperature

state check_temperature:
  transitions:
    checking_loop -> fail
```

```
timeout=420ms -> check_temperature
[temp > avg_temp \ req_temp += 0.2;]
                                    -> check_temperature
// if the velocity went down again, keep speeding up
velocity < req_velocity -> check_velocity
```

4.4 Specification of the FDIR FSM

The FDIR FSM consists of a list of modes and a list of mode transitions. The modes are implicitly defined by the rows of the FDIR specification table. Two additional modes are predefined and added to the list: the nominal mode, which is also the initial one; the stophold mode, which is entered when a recovery procedure fails and cannot be exited (at the moment no reset of the FDIR is considered). Fig. **??** shows a pictorial view of an example FDIR FSM.

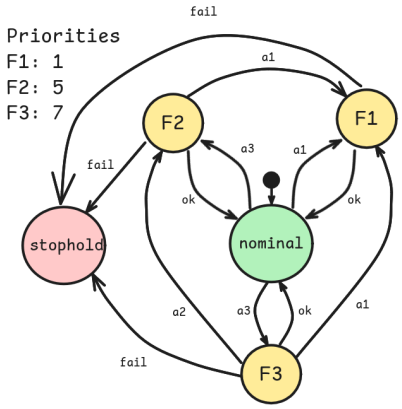

Fig. 5. Top Level FSM example

The specification table assigns a priority to every mode. The nominal mode is given priority 0, while the stophold mode is given a priority higher than any other.

The specification table assigns also a group and to each group a threshold. The FSM maintains a counter for each group, which is increased whenever the FSM enters a mode of that group. Whenever the threshold is reached, the FSM goes to stophold mode. The purpose is to avoid continuous spurious recoveries, which happens when the procedure brings the FDIR to nominal mode, but the problem is not really solved and the same mode is entered again and again.

Overall, the FSM mode transitions are defined as follows. When the FSM receives an alarm from the monitor the next mode is determined by the following rules:

– if the current mode has higher or equal priority than the new one, then the next mode is determined by the current recovery procedure: it remains the current one if not completed, it becomes nominal if completed, or stophold if aborted;
– if the current mode has lower priority of the new one, then the counter associated to the new mode group is increased, and if it did not reached the threshold, the next mode is set to the new mode; otherwise it is set to stophold. If the mode is changed, the counter of the current mode (if the group is different from the new one) is reset.

4.5 Code Generation Process

The code generation is organized into three main parts, reflecting the three different components of the generated code: *monitor generation*, *FSM generation* and *interface generation*.

The *monitor generation* relies on standard runtime verification techniques to generate a deterministic finite-state automaton to monitor a temporal formula (see, e.g., [**?**]). In our implementation, we use the NuRV runtime verifier [**?**] to generate the single monitor code for each property, and then a MonitorHandler is configured to handle these monitors and connect their output to the alarms accepted by the FDIR FSM.

More specifically, the generation of the monitors proceed as follows:

– Preprocess the monitoring conditions to
  • remove time units converting the time constraints into bounds on number of FDIR ticks, and
  • replace functions with fresh variables with the same type of the function return value.
– Generate C monitors with NuRV.
– Generate the MonitorHandler with the glue code to connect the input/output of the monitor with the system controller and the FDIR FSM.

For the *FSM generator*, the code generation relies on OpenGeode which takes in input an SDL description of the FSM. For this reason, the internal representation of the FDIR FSM is created from the table specification and then dumped into a file with the SDL syntax. OpenGeode is then called to generate the corresponding C code.

The final step is to generate the code that provides the interface with the controller. This contains, on one side, all the setters/getters and commands accessible from the FDIR component, and on the other side, the implementation of the *tick* function, which performs the duty cycle of reading inputs, calling the monitor, calling the FDIR FSM to update the FDIR mode, update the state of the recovery procedure in the updated mode and writing outputs.

Running Example The code generation for the interface in our example, could be as follows:

```
// Variables and procedures living in the existing system
extern void restart_power();
extern unsigned char main_power_supply;
// Generated functions ...
void fsm_RI_getsecondary_power_supply(asn1SccTBoolean* value){
    (*value)=secondary_power_supply;
}
//...
```

For the Monitor Handler we follow a clean approach, where we separate the reading of inputs and the checking of the properties (we import a*.h which source files contain the explicit automata for the properties):

```
#include "MonitorHandler.h"
int a3loc = 1; // initial location(state) of the automata.
a3_input_t input_a3;
// ...
void tick_monitors() {
  RV_value val;
  read_inputs(); // Update C monitors with current values.
  val = a3_scalar(&input_a3, NULL, 2, &a3loc);
  if(val == RV_TRUE) fsm_PI_a3(); // send alarm to FSM
  // ...
}
```

The SDL FSM can be viewed from OpenGeode:

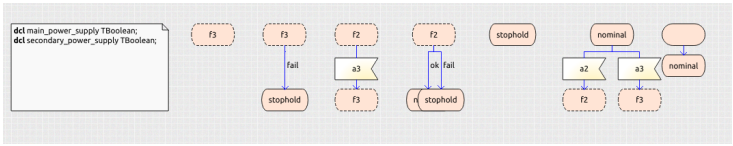

Fig. 6. SDL FSM components of the running example

4.6 Formal Verification

The generation of the SMV model for formal verification follows a similar approach. The SMV model is modular and reflects the hierarchical structure of the FDIR specification. The top level FSM is represented by the main module, while each FDIR mode is represented by a separate module whose behavior represents the corresponding recovery procedure. For lack of space, we omit the details, but the translation follows the semantics specified in the previous sections in a straightforward way.

The generic structure of the SMV model is reported in the following snippet:

```
MODULE main
-- nominal is lowest priority, and stophold highest.
DEFINE priority := [0, ...];
VAR
    current: 0 .. N+1;
    alarm: 0 .. N+1;
    input_fun0: boolean;
    input_fun1: boolean;
    ...
    input0: ...;
    input1: ...;
    ...
    f1: F1(current, alarm, priority, ...);
    f2: F2(current, alarm, priority, ...);
    ...
TRANS -- stophold as trap state.
    current = 1 -> next(current) = 1
-- higher priorities will get to that state, or ok/fail.
TRANS
    priority[current] < priority[alarm] ->
            (next(current) = alarm | next(current) = 1)
-- lower priorities will keep in the same state, or ok/fail.
TRANS
    priority[current] >= priority[alarm] ->
            (next(current) = current | next(current) = 0
            | next(current) = 1 )

MODULE F1(...)
...
```

The `current` variable represents the current mode. The value "0" represents the nominal mode, the value "1" represents the stophold mode, while the other values represent the user-defined modes.

If an alarm is received and the corresponding mode has a priority higher than the current one, the next mode will be set to the new one or to nominal or to failure. If the alarm corresponds to a mode with lower priority is discarded and the next mode is determined by the module implementing the recovery procedure of the current mode.

Finally, each recovery procedure is translated into an SMV module. The transition follows the definition of transition in a straightforward way.

Running Example In the development process, tuning the priorities might not be trivial and could lead to inconsistent definitions. One of the possible errors could be that we set up a priority that never triggers because there is a higher priority failure mode that implies the other property. We can leverage model checking to automate this property: "There is no higher priority failure condition that implies a lower priority failure condition". We might have:

Property 1: $H[0s, 10s] \wedge \neg\text{main} \wedge H[0s, 20s] \wedge \neg\text{secondary}$
Property 2: $secondary\_supply S[10, 10] \neg main\_power\_supply$

If we mistakenly give more priority to Property 2, we find the following warning:

```
-- specification
(((secondary\_power\_supply) S [10,10] (!main_power_supply))
-> ( H [0,10] !main_power_supply &
H [0,20] !secondary_power_supply)) is true
```

Which, in the development process would mean that we can either strengthen the property, remove it, or swap priorities.

5 Heat Pump Controller Case Study

5.1 Description of the System

Innova Engineering is setting up a structured development process for the control software of Innova heat pump systems for heating and air conditioning. As part of this process, the integration of FDIR represents an important asset for reliable control. To avoid entering in proprietary designs of heat pumps, we here use an example that is representative of the most recent products of the company. The example is a cooling/heating system powered by pumps with an exchanger in between, and multiple sensors to monitor the physical dynamics. In order to understand the monitoring properties and recovery procedures, we give a brief introduction to the system.

Fig. **??** gives a pictorial view of the case study with detail of the various components' inputs and outputs. The circuit has two pumps, the master pump which works on nominal conditions, and the redundancy pump, for faulty situations. These pumps make the fluid flow into the cooling/heating module (exchanger). The exchanger modifies the liquid's temperature depending on the requested power, its density and a few more variables. The fluid with the desired temperature will be directed to the hot/cold load, and if no problems were detected, start the cycle again. Later, it will be directed in-between components through pipes with flow sensors, measuring the volume of liquid we transfer per second.

5.2 FDIR Specification

All the information about failure modes is listed in Table **??**, apart from the group thresholds that are omitted for lack of space (note that empty isolation conditions are interpreted as always true). We briefly discuss here some of the most important monitoring conditions and related recovery:

- The sum over differences of pressures is not 0, which could mean that the pump is blocked, or there is a leakage in the pipe system. In this particular example, we experience a rise in pressure (for simplicity, we assume that the pressure drop on the cold load is 0):

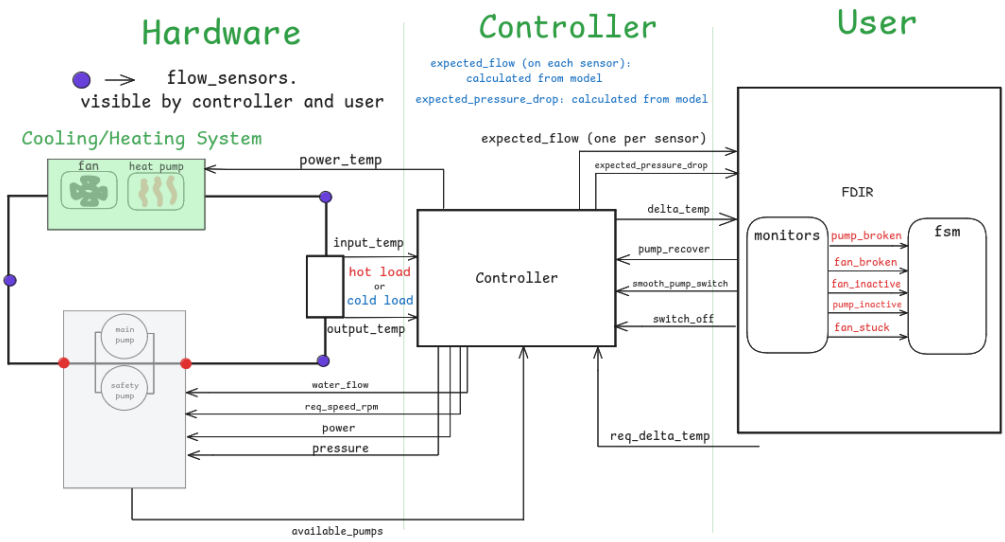

Fig. 7. Heat Pump case study architecture

- The pump: we will model the pressure rise with:

$$\Delta P\,[\text{kPa}] = 5.62 \times 10^{-6} \times \text{pump\_speed}\,[\text{rpm}]^2 - 4.46 \times \text{flow}\,[\text{m}^3/\text{h}]^2$$

- The exchanger: $\Delta P = \left(\frac{\text{flow}}{K_v}\right)^2$, where K_v is a coefficient that takes into account exchanger geometry and fluid type.

To check this condition, we define the following formula:

```
H[0s, 45s] (pressure_diff_pump()+pressure_diff_exchanger())!=0
```

where the functions are provided by the controller interface to compute the above expressions. Furthermore, to isolate the problem we could check, for every flow sensor if $flow\_sensor_i$ is far from the expected, and handle the possible leakage. Otherwise, we can assume that there is a blockage in the pump, and try to unstuck it by repeatedly powering on-off.

– The average flow is too far from the expected value. Possible causes are: the primary pipe has experienced a decrease in flow because of a pump obstruction; or, the channel after the exchanger is not letting the fluid keep the natural flow. One full monitoring property could be the following:

```
H [0s, 30s] abs(expected_avgflow()-avgflow)>2 &
O[0s, 30s] abs(expected_flow(2) - flow_sensor_2) > 1
```

– The redundancy pump has to work since the primary pump broke, at least in a span of 10 seconds.

```
available_pumps[1] S [0s, 10s] !available_pumps[0]
```

fault detection condition	isolation condition	prio	recovery procedure
H[0s, 38s] ! available_pumps[0]	H[0s, 10s] !available_pumps[1]	400	init fail [switch_off();]
		100	init wait_recover [fix_pump(0);] state wait_recover: transitions: timeout=2s -> ok
available_pumps[1] S[0s, 10s] !available_pumps[0]		300	init fail [switch_off();]
H[0s, 45s] (pressure_diff_pump + pressure_diff_exchanger) != 0		300	init power_off fix_timeout: timeout=2m on {power_off, power_on;} state power_off: transitions: flow_sensor_0 < 1 -> ok fix_timeout -> fail [timeout=15s \off_pump();] -> power_on state power_on: transitions: flow_sensor_0 < 1 -> ok fix_timeout -> fail [timeout=15s \on_pump();] -> power_off
H[0ms, 150ms] (temp_diff $\leq 0 \iff$ hot_load)		220	init ok [req_temp_diff := 0;]
H[0s, 30s] abs(expected_avgflow() - avgflow()) > 2	O[0s, 30s] abs(expected_flow(2) - flow_sensor_2) > 1	185	init pump_stucked [unstuck_pump();] state pump_stucked: transitions: [abs(expected_avgflow() - avgflow()) < 1 \ restart_flow();] -> ok [timeout=1m switch_off();] -> fail
	O[0s, 30s] abs(expected_flow(1) - flow_sensor_1) > 2	190	init fail [switch_off();]
	O[0s, 30s] abs(expected_flow(0) - flow_sensor_0) > 1	180	init fixing_exchanger [restart_exchaner();] state fixing_exchanger: transitions: exch_power() > 2.3 -> ok timeout=2m -> fail
H[0ms, 350ms] calculate_fluid(fluid) = fluid_type_table(fluid)		250	init switch_fluid [smooth_switch(fluid);] state switch_fluid: timeout=3s -> fail calculate_fluid(fluid) = fluid_type_table(fluid) -> ok

H [0s, 2m] req_temp_diff - temp_diff < temp_diff_thresh	H[0m, 2m] (fan_req_speed > 0 & fan_speed > 0 & abs(fan_req_speed - fan_speed) > fan_thresh)	160	init recover_fan [
			initial_req := fan_req_speed;
			fan_req_speed := 0;
			]
			state recover_fan:
			[timeout 1m \fan_req_speed
			+= fan_rate] -> recover_fan
			abs(fan_req_speed - fan_speed) < 3 -> ok

	H[0m, 4m] (fan_speed = 0 & fan_req_speed != 0)	160	init fail [switch_off();]
	H[0m, 2m] power_temp_req > MAX_power	200	init ok [power_temp_req := MAX_power;]

Table 1. FDIR specification of the case study

Finally, note the combination of detection and isolation conditions with priorities. For example, in the case of the flow sensors, we detect a fault when the average flow diff has been greater than 2 for the last 30s. We then identify which sensor has a problem by checking each sensor in the last 30s, but since are flow sensors in different parts of the circuits, they are given different priorities, based on their criticality.

5.3 Verification Results

The specification table described in the previous section is used to generate the code of the heat pump controller. The code of the hierarchical FSM combining

the FDIR modes and recovery procedures results in C file with 2277 lines of code, without counting the monitors and the controller interface.

In order to check the correctness of this FDIR, we used nuXmv for model checking the generated SMV model, and various tests to verify the generated code. These two verifying techniques are complementary, as the model checking of SMV formally validates that what we wrote in the table specification is what we had in mind, while the testing verifies the actual program execution.

For testing, we set up a simple testing framework where input values as expected output are specified in csv files, with the possibility to inject faults. Tests in C are automatically generated to execute the tests.

We tested each FDIR mode in the presence of no other alarm to verify the correct flow of the state machine in each FDIR mode in isolation. Complementary, we also wrote tests, where higher and lower priority alarms are triggered, and we ignore/react to these information. Since we implemented 12 different properties, we covered a total of 36 different scenarios.

As for model checking, we checked 1) each failure mode transition 2) mode transitions with alarms of different priorities, 3) absence of infinite loops, and 4) reachability of group thresholds.

These kinds of checks are particularly useful for early verification and validation, to find issues in the table specification. An example of issues in an earlier version of the table was found checking a recovery procedure in isolation. The property was $G(alarm = 4) \rightarrow F(current = 0 \,|\, current = 1)$. A counterexample showed that there was an infinite loop moving between power_off and power_on. These problems are solved by introduction timeout groups. We also noticed possible infinite loops when we have a self loop with a condition, and the timeout that keeps track of this loop is in the lowest priority (since the higher priority may always be true, and always keep looping).

6 Conclusions and Future Works

In this paper, we presented a specification-driven approach to the automatic generation of embedded code for fault detection, isolation, and recovery. The proposed approach translates high-level formal specification of monitoring conditions and recovery procedures into platform-specific embedded code. Additionally, model checking techniques provide a means to validate and verify the specified FDIR logic, strengthening its reliability. We demonstrated the approach through a case study in the energy domain.

While the proposed approach significantly reduces manual coding effort by automating the generation of FDIR code from high-level specifications, it still requires to define fault detection conditions and recovery logic in formal notations. Future work will aim to further streamline this process through increased automation. Other directions for future work include optimizing the generated code to meet requirements on timing and memory bounds of specific target platforms, and connecting the proposed methodology to preliminary hazard analysis phases and to system-level analysis of failure modes and effects.