FAILURE MODE AND EFFECTS ANALYSIS (FMEA) AND MODEL-CHECKING OF SOFTWARE FOR EMBEDDED SYSTEMS BY SEQUENTIAL SCHEDULING OF VECTORS OF LOGIC-LABELLED FINITE-STATE MACHINES

V. ESTIVILL-CASTRO, R. HEXEL*, D.A. ROSENBLUETH†*

**Griffith University, Brisbane, Australia. {v.estivill-castro,r.hexel}@griffith.edu.au,*
†Universidad Nacional Autónoma de Mexico, México City, Mexico. drosenbl@unam.mx

Keywords: Hazard identification, Software safety, Risk assessment, Safety and systems engineering, Safety cases.

Abstract

Model-Driven Development (MDD) has proven to be a very powerful tool to produce software for embedded systems that control sophisticated equipment. It is therefore even more critical that such software be verified to be correct and to clearly understand what the safety implications of potential failures in sensors, actuators or faults of the software itself are. Using vectors of logic-labelled finite state machines, a clear semantics is obtained as well as executable models that provide the benefits that MDD promises. Since we can perform effective model-checking on these models, we show in this paper that we can use this to systematize and automate the failure mode and effect analysis of systems with embedded software. We illustrate this with two ubiquitous examples in the literature of model-checking for software in embedded systems.

1 Introduction

More and more, software for systems (in particular embedded systems) is being developed under a Model-Driven Development (MDD) approach. Model-Driven Engineering provides the capacity to describe behaviour at a high level and allows direct implementation on many platforms. Software production using MDD presents numerous advantages. Hence, more and more systems are using software that was developed under such an approach. This makes it far more critical that such software be correct by default. Ensuring that the software is free of faults is critical, because software failure can lead to catastrophic failure of machines and equipment driven by such software. Avoiding this is central to system safety.

A very successful tool for model-driven engineering of embedded systems is finite-state machines (FSMs) whose transitions are labelled with expressions of a common-sense logic [2]. These are remarkably expressive when compared to other paradigms for modelling behaviour, such as Petri Nets, Behavior Trees, and even standard FSMs, e.g. those of executable UML [8] or StateWorks [12]. The use of efficient and practical model-checking techniques, however, was previously limited. The reason is that only a single, sequential FSM [4] could be efficiently model-checked, whereas systems are usually composed of various components exhibiting concurrent behaviours.

We have recently shown [3], nevertheless, that models composed of multiple finite-state machines can be structured as a vector whose execution uses a round-robin, sequential, off-line schedule. This enables efficient model-checking of the requirements. This extension also includes an extension to the implementation that builds Kripke structures for an individual FSM to one that derives rules for a collective of sequential finite state machines (that operate following a deterministic sequence). Moreover, the very interpreter that actually implements and runs these rules also constructs the corresponding Kripke structure. As a result, formal verification of correctness properties using standard tools such as NuSMV is possible.

In this paper, we show how this technology can be applied to carry out *failure mode and effects analysis* (FMEA) directly, automatically (and with a mathematical foundation) from the software model. That is, the model-checking aspect completely verifies the desirable properties. The model (having several components) is shown to be correct. Importantly, we can systematically study the effects of hypothetical defects or failures in one or more components. Such hypothetical situations are reflected as properties that no longer hold.

Moreover, this analysis not only carries out FMEA for the software; we describe here how the analysis can be carried out for other parts or participants in the system. That is, we can model (using the vector of logic-labelled finite-state machines) sensors, actuators, and operators outside the software itself. For this enlarged system, we can investigate which properties are impacted by unanticipated or faulty behaviour of such sensors, actuators, or human operators. The MDD approach and its interpretation enable also the validation by simulation of the FMEA.

We illustrate this approach with two case studies widely used in the software engineering and model-checking literature (the mining pump and the industrial press). Correct behaviour of these systems is crucial due to their safety requirements.

2 Logic-Labelled Finite-State Machines

Finite-state machines (FSMs) are a widely used formal model for system behaviour, particularly the dynamics of a software system. Typically, a finite state machine is composed of a set S of states, one of which is designated an initial state s_0. The behaviour is specified by the transitions that shift the machine form a current (source) state to a new (target) state. We consider here Logic-Labelled Machines; that is, for each source state s_i there is a logic theory T_i and a list of transitions $<t_{i1}, t_{i2},...,t_{in}>$. Each transition t_{ij} is labelled with a proposition p_{ij} of the logic and a target state s_{ij}. It is important to highlight that the outgoing transitions of a state are represented as a list, since the semantics of the FSM consists of evaluating each proposition in sequence (determining if in the theory T_i the proposition is true). When a proposition is true, the transition fires and the current state changes to the corresponding target state.

States themselves have three sections. An *OnEntry* section, an *OnExit* section, and an *Internal* section. The *OnEntry* section and the *OnExit* section are always executed exactly once for each state. The *OnEntry* section is executed as soon as a machine makes such a state the current state, while the *OnExit* section is executed on departure from a state. Thus, the *OnExit* section of a transition source state is executed before the *OnEntry* section of the target state. The *Internal* section is only executed when no transition fires, i.e. the list of the outgoing transitions of the current state is exhausted without any proposition of the transition list evaluating to true. After the *Internal* section is executed, the FSM resumes evaluating the list of transitions from the beginning.

Figure 1 shows an example of such a FSM. The language for transitions is expressions in simple C (C without control structures, but including function calls and assignment), while the language for a state section uses statements in simple C.

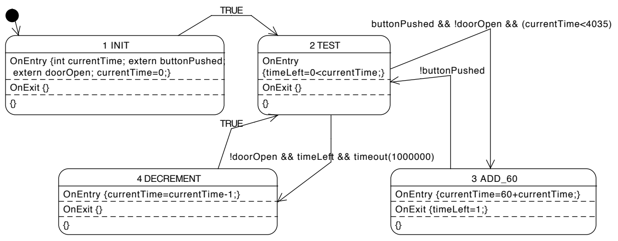

TRUE
buttonPushed && !doorOpen && (currentTime<4035)
1 INIT
OnEntry {int currentTime; extern buttonPushed; extern doorOpen; currentTime=0;}
OnExit {}
{}
2 TEST
OnEntry {timeLeft=0<currentTime;}
OnExit {}
{}
!buttonPushed
TRUE
!doorOpen && timeLeft && timeout(1000000)
4 DECREMENT
OnEntry {currentTime=currentTime-1;}
OnExit {}
{}
3 ADD_60
OnEntry {currentTime=60+currentTime;}
OnExit {timeLeft=1;}
{}

Figure 1. FSM for the timer of a Microwave.

In Fig. 1, for example, the initial state is state `1 INIT`. There is a transition from state `2 TEST` to state `3 ADD_60` whose proposition is

```
buttonPushed && !doorOpen && (currentTime < 4035)
```

that indicates the timer should change state to a state that adds 60 seconds to `currentTime` (the *OnEntry* section of the `3 ADD_60` state) if the button has been pushed, the door is not opened, and the current time is below the maximum value (so that a further 60 seconds would not overflow its register).

3 Vectors of logic-labelled FSMs.

The challenge, however, is that although in theory a system can be represented by a single logic-labelled finite-state machine, such a model is usually extremely hard to comprehend or design. It is much more manageable to build a system of conceptually separate, interacting components. Our case studies will illustrate this. For an immediate example, we note that Fig.1 is just the timer of a microwave, but there are other components like the light (which has two states, on when the door is open or the beam is cooking, or in the state of off when the door is closed and cooking is not going on). Similarly, there is a bell that becomes armed when cooking starts and rings when cooking completes.

If one considers all these components as concurrent subsystems, then the challenge is that the possible states of the composed system is the Cartesian product of the space of states of the components. Any part of the system may change state without the other parts acting. This quickly becomes an unmanageable state space (in order to perform model-checking). Note that most likely the designers and users of the system does not need, require or desire this wide capacity for arbitrary behaviour. And in fact, in the context of system safety, what is desired is a precise, and possibly unique path or significantly constrained behaviour.

We use an approach inspired by the time-triggered architecture for scaling down this complexity. The set of logic-labelled FSMs that constitute a system is also represented as a list. Commencing from the first FSM on the list, the semantics of the vector-model is that, of the current machine, only one ringlet is executed. That is, on the current machine, the analysis of its current state is performed. For the initial ringlet of the state, the *OnEntry* section proceeds, followed by the evaluation of the list of transitions until

- one transition fires and the *OnExit* section is executed or
- none of the transitions fires and the *Internal* section is executed.

Once this happens, the execution of the vector moves to the next FSM in the list (in circular fashion, the last FSM is followed by the first).

This actually converts the execution of concurrent FSMs into a single thread with far fewer states and deterministic execution, enabling model-checking on a complex system.

Such a vector of deterministically executing FSMs also greatly simplifies communication, as all synchronisation points are now deterministic as well. In terms of semantics, if a variable has no declaration, it is shared by the vector, but never modified by the environment (so in the language of model-checking, the Kripke structure is deterministic, and only simple C statements in the FSMs can modify its value).

A variable in a FSM can be further restricted to be a local variable and thus, not visible to any other FSM in the vector (for example, the declaration `int currentTime` in state `1 INIT` of Fig. 1). Variables declared `extern` are not only common to all FSMs but can also be modified from outside the software. In the example of Fig. 1, we have `extern buttonPushed` because it is not the software of the timer which set the variable `buttonPushed` to true or false, but the operator of the microwave by pressing the button (or the button by being faulty). In the language of model-checking, such external variables lead to non-deterministic states in the Kripke structure and so is the case in our interpreter and generator of Kripke structures for the model checker NuSMV. Thus, our Kripke structures correctly model the software being in any possible system state (for example, the microwave could be turned on, with the user pushing the button, or it could also be turned on due to a faulty button). For safety, it is critical that the model-checking of the software be performed with all possible combinations of external variables the software does not have control over. Importantly, in our interpreter, external variables are read only once, at the commencement of the evaluation of a ringlet. This prevents inconsistencies due to changes within the environment once a Kripke state has commenced. Moreover, this ensures that system safety properties do not depend on the speed the software runs relative to sensor perception. That is, if we prove our models correct (with formal model-checking), such correctness is not subject to the software running fast enough. In the example of the microwave, while the user may push and release the button too quickly to be noticed, such input cannot trigger unspecified system behaviour or falsify a property of the behaviour. Even if the user pushes and releases the button too quickly to be detected, we can formally prove that if the button is held for long enough to process a minimum number of Kripke states, correct behaviour will ensue.

4 Case studies

We show here two case studies that first show how the techniques described above are used to formally prove (using model-checking) safety properties of the software (beyond what previously has been reported in the literature). Then we proceed to describe how the models enable *failure mode and effects analysis.*

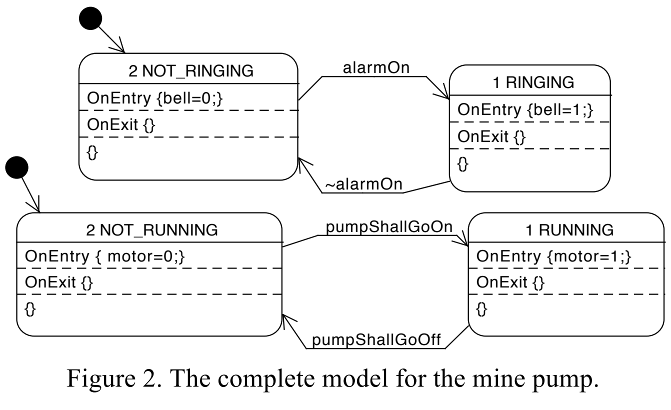


Figure 2. The complete model for the mine pump.

4.1 The Mining Pump

The mining pump is a case widely discussed in the literature [4, 9, 10] where software is controlling a safety-critical system. The pump prevents a mineshaft from flooding, but must not run if flammable gas concentration is high. We follow Burns and Lister [1] where formality is provided. We use this case study as it enables the presentation of our methodology, for constructing the model, for performing model checking, for performing *failure mode and effects analysis* (FMEA) [11], and for directly executing the models on a platform (thus, fulfilling the promise of model-driven engineering). This case study is also illustrative of the power of using FSMs with transitions labelled by a question to a common-sense logic such as DPL [2].
We summarise here the development of the model. Details can be found elsewhere [3]. The initial model for this system appears in Figure 2. It consists of a vector of two logic-labelled FSMs, each with its associated logic.

```
%Alarm.d
name{ALARM}.
input{CO2SensorHigh}.
input{airFlowLow}.
A0: {} =>  ~alarmOn.
A1: CO2SensorHigh =>  alarmOn. A1>A0.
A2: airFlowLow =>  alarmOn. A2>A0.
output{b alarmOn,"alarmOn"}.
```

Figure 3. The logic for the FSM for the alarm.

```
name{MINEPUMP}.
input{lowWaterSensorOn}.  input{highWaterSensorOn}.
input{operatorButtonOn}.
input{methaneSensorHigh}.  input{indicateOn}.  input{indicateOff}.

P0: {} =>  ~pumpShallGoOn.
P1: highWaterSensorOn =>  pumpShallGoOn.
P1>P0.
P2: lowWaterSensorOn =>  ~pumpShallGoOn.
P2>P1.
P3:
{~lowWaterSensorOn,~highWaterSensorOn,operatorButtonOn}=>
pumpShallGoOn.                  P3>P2. P3>P0.
P4:
{~lowWaterSensorOn,~highWaterSensorOn,~operatorButtonOn}=
> ~pumpShallGoOn.                  P4>P3.
P5: indicateOn => pumpShallGoOn.
P5>P2. P5>P4. P5>P0.
P6: indicateOff => ~pumpShallGoOn.
P6>P5.
P7: methaneSensorHigh => ~pumpShallGoOn.
P7>P5. P7>P3. P7>P1.

N0: {} =>  ~pumpShallGoOff.
N1: {~indicateOn,lowWaterSensorOn} =>  pumpShallGoOff.
N1>N0.
N2:
{~indicateOn,~lowWaterSensorOn,~highWaterSensorOn,~operato
rButtonOn}=> pumpShallGoOff.      N2>N0.
N3: indicateOff => pumpShallGoOff.
N3>N0.
N4: methaneSensorHigh => pumpShallGoOff.
N4>N0.

output{b pumpShallGoOn,"pumpShallGoOn"}.  output{b
pumpShallGoOff,"pumpShallGoOff"}.
```

Figure 4. The logic for the pump engine.

The logic appears in Figure and 4: the FSMs for the alarm will move from the state of *not ringing* to *ringing* if the logic can determine the predicate `alarmOn`. The logic would like to have information about whether the CO_2 level is high or the airflow is low (that is the role of the `input` clauses). The logic will attempt to establish the predicate `alarmOn` (this is the role of the `output` clause). The "=>" operator is read as "usually" (unlike implication in the sense of propositional logic); for example, rule `A0` says that usually the alarm is not on, but rule `A1` says that if the CO_2 level is high, the logic shall recommend the alarm be on; and rule `A1` takes precedence over `A0` (this is the role of the precedence operator ">"). In practice, modelling with FSMs and non-monotonic logic results in very concise and transparent models. For example, Fig. 4 is the complete model for the mine pump. By comparison, an equivalent model using Behavior Trees takes several A4 pages. A video of the model in operation and validating the 3-state operator switch vs. the 2-state operator switch appears in youtu.be/y4muLP0jA8U.

The compilation of the logics into simple expressions and the analysis of safety properties using model checking reveal that the switch of the supervisor must be a 3-state switch (*on*, *off* and *inactive*). Once the logics are compiled, the complete, correct model (proven by model-checking the derived Kripke structure generated as described earlier) appears in Fig. 5.

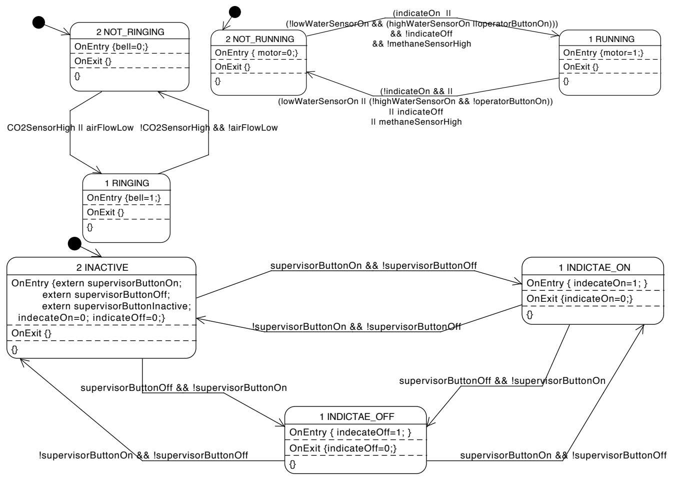

2 NOT_RINGING
OnEntry {bell=0;}
OnExit {}
{}
CO2SensorHigh || airFlowLow
!CO2SensorHigh && !airFlowLow
1 RINGING
OnEntry {bell=1;}
OnExit {}
{}
2 NOT_RUNNING
OnEntry { motor=0;}
OnExit {}
{}
(indicateOn ||
(!lowWaterSensorOn && (highWaterSensorOn ||operatorButtonOn)))
&& !indicateOff
&& !methaneSensorHigh
1 RUNNING
OnEntry {motor=1;}
OnExit {}
{}
(!indicateOn && ||
(lowWaterSensorOn || (!highWaterSensorOn && !operatorButtonOn))
|| indicateOff
|| methaneSensorHigh
2 INACTIVE
OnEntry {extern supervisorButtonOn;
extern supervisorButtonOff;
extern supervisorButtonInactive;
indecateOn=0; indicateOff=0;}
OnExit {}
{}
supervisorButtonOn && !supervisorButtonOff
!supervisorButtonOn && !supervisorButtonOff
1 INDICTAE_ON
OnEntry { indecateOn=1; }
OnExit {indicateOn=0;}
{}
supervisorButtonOff && !supervisorButtonOn
supervisorButtonOff && !supervisorButtonOn
1 INDICTAE_OFF
OnEntry { indecateOff=1; }
OnExit {indicateOff=0;}
{}
!supervisorButtonOn && !supervisorButtonOff
supervisorButtonOn && !supervisorButtonOff

Figure 5. Complete and correct model of the mine pump with all expressions compiled to simple C.

In this model, we can formally verify that the software satisfies several safety properties, far more properties than anywhere before in the literature.

- Property-1 *"If CO_2 is high, the alarm must ring."*
- Property-2 *"If air flow is low, the alarm must ring."*
- Property-3 *"If methane levels are high, the pump must be turned off."*
- Property-4 *"If the supervisor switches off when running, the pump will be turned off."*
- Property-5 *"If the operator turns her switch off when the pump is running and the water level is neither low nor high, then the pump motor goes off."*
- Property-6 *"The pump is turned on when the water is above the high water sensor (and the low-water sensor's signal is consistent with this), unless the supervisor turns it off or methane levels are high."*
- Property-7 *"If the supervisor sets the switch to inactive and the pump is running when the water is not above the high water sensor and the low-water sensor indicates a low level, the pump turns off."*
- Property-8 *"If there is low methane, low water, and the pump is not running, but the supervisor puts the switch to on, then the pump is turned on."*

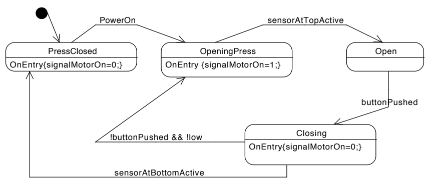
We stress this point as models in previous research articles actually fail some of these properties (or incorrectly suggest that a 2-way switch for the operator is sufficient). Moreover, we can not only model and simulate the software but the other components of the system as well, and perform a *failure mode and effects analysis*. We can efficiently generate tables such as the one shown below, which was obtained by injecting faults through altering the FMSs that represent the sensor and/or actuator, and then re-running the model checker. A similar table can be obtained for the properties that fail if multiple sensors and/or actuators fail simultaneously (e.g. with common-mode failures that cannot just be derived from Table 1).

Failures	Consequences							
	Property that fails							
	1	2	3	4	5	6	7	8
CO2-sensor stuck high								
CO2-sensor stuck low	X							
Airflow sensor stuck high		X						
Airflow sensor stuck low								
Bell stuck ringing								
Bell stuck not ringing	X	X						
Supervisor button stuck in on				X			X	
Supervisor button stuck in off						X	X	X
Operator button stuck in on					X			
Operator button stuck in off							X	
Methane sensor stuck in high						X		X
Methane sensor stuck in low			X					
(High water) sensor stuck in on					X		X	
(High water) sensor stuck in off						X	X	
(Low water) sensor stuck in on						X		
(Low water) sensor stuck in off					X	X		X
Motor stuck running			X	X	X		X	
Motor stuck not running						X		X

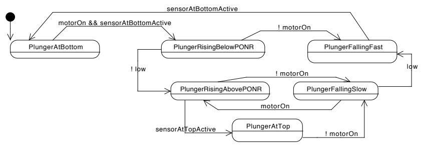
Table 1. FMEA Table Level 1 for the mine pump case study.

4.2 The Industrial Press

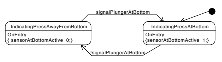
The industrial metal press has been also widely studied in the literature of model-checking for failure analysis [5, 6, 7]. In this system, a plunger is initially resting at the bottom with the motor off. When power is supplied, the controller turns the motor on, causing the plunger to rise. When at the top, the plunger shall be held there until the operator pushes and holds the down the button. This causes the controller to turn the motor off and the plunger to fall. If the operator releases the



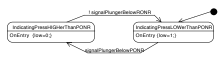
(a) States for the Controller.



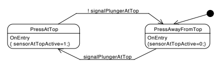
(b) States for the Plunger.


signalPlungerAtBottom
IndicatingPressAwayFromBottom
OnEntry
{ sensorAtBottomActive=0;}
IndicatingPressAtBottom
OnEntry
{sensorAtBottomActive=1;}
!signalPlungerAtBottom

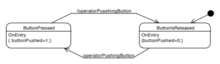
(c) States of the Bottom Sensor.



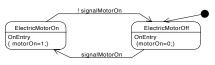
(d) States of the PONR Sensor.



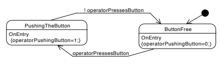
(e) States of the Top Sensor.



(f) States of the Button.



(g) States of the Electric Motor.



(h) States of the Operator.

Figure 6. Complete Model of the Industrial Press.

button while the plunger is falling slowly, above a point of no return (PONR), the controlling software turns the motor on again, causing the plunger to start rising again, without reaching the bottom of the press. However, if the plunger is falling fast, below the point of no return, then the controlling software leaves the motor off until the plunger reaches the bottom. This is the main critical safety feature, as turning the motor on with too much inertia on the plunger results in catastrophic scenarios. When the plunger reaches the bottom, the software must receive a signal to turn the motor on, so the plunger rises again.

We emphasize that the summary above of the behaviour of this system may differ from other descriptions in the literature, as these differ among themselves when one analyzes the details. For example, the On/Off button is modelled inaccurately and the infra-red link described in the original source is not described in recent FMEA analysis using design behaviour trees. This results in an unsafe cyclical behaviour of the system for that model. Namely, it is possible for the operator to keep the button pushed and the power on, enabling the motor to raise the plunger. As a result, the motor goes off bringing the plunger down, and then as the power is on (no switch on the motor), the motor goes on again raising the plunger immediately. We have demonstrated this fault in such model in simulation and by running the model in two platforms (illustrating this defect in a video http://youtu.be/blUpMdH14pM), and also the corresponding correction. Such a correction appears as part of the properties we can formally verify about the corrected model.

- Property-1 *"If the operator is not pushing the button and the plunger is at the top, the motor should remain on"*.
- Property-2 *"If the plunger is falling below the PONR, a state modeled by the plunger falling fast, then the motor should remain off."*
- Property-3 *"If the plunger is falling above the PONR, a state modeled by falling slow, and the operator releases the button, the motor should turn on, before the plunger changes state."*
- Property-4 *"Once the plunger is down, a new signal is needed to turn the motor on and raise the plunger again."*

Our approach again allows carrying out the *failure mode and effects analysis* (FMEA) by systematically injecting failures in the FSMs that model the sensors and the actuators of the system. Again, we illustrate this by obtaining a table of properties that fail with one component failing. Clearly, the exercise can be repeated for two components failing simultaneously (or 3 or more). That is, our software generates the Kripke structure for input for NuSMV when presented with the vector of FSMs with one faulty, and the model-checker highlights which of the properties no longer holds.

Failures	Consequences			
	Property that fails			
	1	2	3	4
Bottom sensor stuck indicating press away from bottom				X
Bottom sensor stuck indicating press at bottom				
PONR sensor stuck on above PONR		X		
PONR sensor stuck on below PONR			X	
Top sensor stuck indicating press away from top	X			
Top sensor stuck indicating press at top				
Operator button stuck on pressed	X			
Operator button stuck on released				
Motor fails, leaves motor stuck on running				X
Motor fails, leaves motor stuck on off	X			X
Power switch button stuck to supply power				X
Power switch button stuck to no power	X	X	X	X

Table 2. FMEA table at level 1 for the industrial press.

In Figure we have modelled actuators like the motor (the FSM in Figure g) and thus can introduce a failure to the system as a faulty motor. We have also modelled all the sensors, and even the operator (Figure h). Thus, one can even analyse behaviour by the human operator that is not compliant. We emphasize here we are not aiming at being comprehensive. These are illustrative case studies. For example, we could also model other failures. Like a bottom sensor which, with certain frequency/probability, indicates incorrectly the position of the plunger and analyze the properties that no longer hold regarding the correctness of the system. Our point here is that the FMEA analysis is significantly automated for the safety analyst. We also have a formal methodology that provides clear and sound evidence of the correctness and reliability of the software. In particular, the path to establish the criticality of each sensor/actuator with respect to safety, to ensure a minimum standard of to mitigate a risk, and perhaps introduce further redundancy or safety checks becomes systematic.

Furthermore, we do not make any assumptions about initialisation of variables, or the state of external inputs when the FSMs start. This allows for a more rigorous mathematical proof of safety properties without any implied "common sense" assumptions. For example, the LTL formula to verify Property 1 above was defined as

```
LTLSPEC
G ( (buttonPushed=0 & sensorAtTopActive=1) ->
signalMotorOn=1
)
```

in [4: page 1248], making the assumption that the software would already have been initialised. In fact, this property is false in the initial state of the Kripke structure. For our models, we do not make such implicit assumptions, but express this in the `NuSMV` model-checking language. For example, the property is true if the FSM for the controller (Figure a), by indicating that state 1 `OpeningPress` must have been reached:

```
LTLSPEC
G ( (buttonPushed=0 & sensorAtTopActive=1 & pc =
M0S1R0) -> X(signalMotorOn=1)
)
```

4 Conclusions

Our approach here contrasts with the Behavior Tree approach, where several concurrent threads and channels of communication occur within the Behavior Tree, resulting in computations too complex to perform any sort of formal model checking without a large number of simplifying assumptions. None of our verifications required more than a few seconds, although in some cases, the generated Kripke structures result in files of several Megabytes in size (by contrast, comparable Kripke structures for independent execution would be in the Gigabyte or even Terabyte range).

Moreover, previous research of this type has assumed that any update of internal variables always takes precedence over external events (arguing that the software runs much faster than the possibility of a user pressing and releasing a button, but there is also an admission that this is risky [5] and their model checking is not sound). We do not have to make these types of assumptions. We establish very clearly the point in the ringlet of a finite-state machine that a snapshot of the environment is taken, and do not make any assumptions about the speed and timing of sensor updates. Our models exhibit deterministic behaviour even if sensors are updated much faster than the software may be able to execute.

5 Acknowledgements

We thank members of the Mi-Pal team at Griffith University for the implementation of many aspects of the infrastructure that enable vectors of logic-labelled FSM to be designed, interpreted and translated into Kripke structures for `NuSMV`. We thank the Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS), at the Universidad Nacional Autónoma de México (UNAM) for providing their facilities.

References

[1] A. Burnsand A. Lister, "A frame work for building dependable systems," *Computer J.,* vol. **34**, no. 2, pp. 173–181, 1991.

[2] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Non-monotonic reasoning for requirements engineering," in Proc. 5th Int. Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). Athens, Greece: SciTePress — Science and Technology Publications (Portugal), 22-24 July 2010, pp. 68–77.

[3] V. Estivill-Castro, R. Hexel and D. A. Rosenblueth, "Efficient Modeling of Enbedded Software Systems and Their Fromal Verification", to appear.

[4] V. Estivill-Castro and D. A. Rosenblueth, "Model-checking of transition-labeled finite-state machines," in *Proc. 2011 Int. Conf on Advanced Software Engineering & Its Applications,* Communications in Computers and Information Science, T.-H. Kim et al., Eds., vol. **257**. Springer Verlag, 2011, p. 61.

[5] L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay, "Experience with fault injection experiments for FMEA," *Software, Practice and Experience,* vol. **41**, no. 11, pp. 1233–1258, 2011.

[6] T. Mahmood and E. Kazmierczak, "A knowledge-based approach for safety analysis using system interactions," in *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific,* Dec. 2006, pp. 445 –452.

[7] T. McDermid, J.and Kelly, "Industrial press: Safety case," High Integrity Systems Engineering Group, University of York, Tech. Rep., 1996.

[8] S. J. Mellor and M. Balcer, Executable UML: A foundation for *model-driven architecture*. Reading, MA: Addison-Wesley Publishing Co., 2002.

[9] A. Shrivastava, M. L. V., and B. Randell, "The duality of fault-tolerant system structures," *Software — Practice and Experience,* vol. 23, no. 7, pp. 773–798, 1993.

[10] M. Sloman and J. Kramer, *Distributed systems and computer networks*. Hertfordshire, UK: Prentice Hall 1987.

[11] D. J. Reifer, "Software failure modes and effects analysis," *Reliability, IEEE Transactions on,* vol. **R-28**, no. 3, pp. 247 –249, Aug. 1979.

[12] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach*. NY: CRC Press, 2006.