Efficient Model Checkign and FMEA Analysis with Deterministic Scheduling of Transition-Labeled Finite-State Machines

Vladimir Estivill-Castro
School of ICT,
Griffith University,
Nathan 4111, Australia
Email: v.estivill-castro@Griffith.edu.au

René Hexel
School of ICT,
Griffith University,
Nathan 4111, Australia
Email: r.hexel@Griffith.edu.au

David A. Rosenblueth
IIMAS,
UNAM
01000 México D.F, México
Email: drosenbl@unam.mx

***Abstract*—A very successful tool for model-driven engineering of embedded systems is finite-state machines whose transitions are labeled with expressions of a common-sense logic. The deployment of models to different platforms and different programming languages makes it more imperative to confirm that the models are correct. However, systems are usually composed of concurrent behaviours, which complicates the potential use of model-checking technology. We structure models composed of several finite-state machines into a vector whose execution is a round-robin sequential off-line schedule. This enables model-checking of the requirements. We illustrate this with two case studies widely discussed in the literature. The models can be executed on diverse platforms, and we utilise the same interpreter to generate the corresponding Kripke structure suitable for verification with tools such as `NuSMV`.**

***Keywords*-Model-driven engineering, embedded software, Model-checking, sequential finite-state machines.**

I. Introduction

Modelling the behaviour of a software at a high-level is becoming far more prevalent. Today, high-level specifications of behaviours are largely represented by models using mechanisms such as finite-state machines or Behavior Trees. Such modelling fits well the model-driven engineering (MDE) agenda [1]. The direct interpretation of the models minimises faults (with respect to traditional software development and deployment that translates requirements into implementations). However, this is spurious if the model is not correct in the first place. Hence the critical importance of verifying the models. In many of the applications of component-based systems (where the behaviour of each component is specified by a finite-state machine), the space of states for the system is extremely large and model-checking techniques face a scalability challenge.

We argue here that the required system behaviour is the result of each finite-state machine (FSM) behaving properly in just one schedule that ensures each component operates correctly and timely with respect to the others, so that ensuring that all possible schedules are correct is not necessary. We thus obtain concise (Kripke) structures to which one can apply state-of-the-art model-checking tools (`NuSMV`).

II. The modelling tool

A. Finite state machines with transitions labeled by common sense logic

The modelling tool presented here is the combination of two fundamental paradigms. First, the reactive nature of deterministic FSMs. This is complemented with the reasoning capability of mechanisms of non-monotonic inference and common sense. A FSM consists of a set S of states and a transition partial function (or table) $T\colon S \times E \to S$. There is a distinguished state $s_0 \in S$, named the initial state. The set E is a set of Boolean expressions.

The standard general description of the semantics for $T(s_i, e_t) = s_j$ is that when the machine is in state $s_i \in S$ and the expression e_t evaluates to true, the machine will move to the state s_j. However, this requires that if $T(s_i, e_t)$ and $T(s_i, e_s)$ are defined and they are different, then e_t and e_s never be true simultaneously (for any $t \neq s$).

We simplify the burden for the behaviour designer by making the projection of T on each state a sequence instead. That is, $T(s_i, e_t) = s_j$ will cause a transition to state s_j if e_t evaluates to true and no previous expression ($e_s \,\forall s < t$) in the sequence $T(s_i, \cdot)$ evaluates to true. Note that while this is simply syntactic sugar, it does make the task of the behaviour designer a lot simpler.

States in a FSM model a space where actions take place. These actions are grouped into three sections. An **OnEntry** section is executed upon arrival to a new state, while actions in the **OnExit** section are executed as the machine departs when a transition is fired. While actions in these two sections are executed only once, the third section contains internal actions that are executed every time if no transition has fired. When the internal actions are completed, the machine returns to evaluate the sequence of expressions that label transitions out of the state and the cycle is repeated. We refer to one pass over the cycle as a *ringlet*. An operational semantics of such execution of a single machine has been already provided [2].

It is important to note that the transitions are labeled with expressions of a non-monotonic logic. We emphasise here that we do not favour any particular logic, the only requirement is that evaluation of such expressions is sequential, as in a function call. That is, their evaluation constitutes a theorem-proving query to a (single threaded) inference engine. However, we will use here examples from DPL [3] since, in its implementation, queries in this logic always terminate. Also, the current implementation of DPL allows rules to be pre-compiled to produce equivalent expressions that can directly be passed to a `C/C++` compiler (`+c` option) or evaluated by the embedded `C/C++` proof engine [4] (`+C` option). Moreover, we have extended the modelling language so that we not only communicate with commands to actuators, but integer variables can be used to receive updates from sensors. This language is a subset of the expressions and statements of the `C` programming language. Actions that are not statements of the programming language are direct messages to actuators using essentially the syntax of a function call.

B. Individual, sequential FSMs as components

Frameworks for FSMs constitute two basic tools: (1) a development environment (where models can be drawn, syntactically verified, and communication patterns analysed) and (2) a run-time environment (where models are simulated or executed, debugged and traced, and executions are logged). Moreover, the run-time scheduling of several models of behaviour (each represented by a FSM) can be made deterministic [5]. The important aspect of deterministic scheduling is the possibility of performing model-checking [5]. It has been suggested that sequential pre-computed scheduling would cope with the combinatorial state explosion and complexity problems of mixed synchronous/asynchronous systems that include data flow and task-control derived from using several FSM models [5], but actual integration with model-checking technology was left for further work [5].

Here we propose that a sequence of FSMs with transitions labeled by Boolean expressions are interpreted in a single thread. By providing this semantics, the construction of a Kripke structure effectively becomes the construction of a Kripke structure for a sequential program. A sequence of FSMs is executed in a circular round-robin fashion in the order of the sequence. Each machine runs a *ringlet* [2] of its current state. That is, if the current state is different from the previous state, the **OnEntry** part of the current state is run. Thereafter, a snapshot is taken of all the external and shared variables (see below), and the machine progresses with the evaluation of transition expressions for the current state. If a transitions fires the **OnExit** part is executed, followed by passing control to the next FSM in the sequence (with the first machine considered to follow the last one, completing the circle). If no transition fires, the internal actions of the current state are executed and control passes to the next machine in the sequence in the same way.

C. Generating the Kripke structure for concurrent machines

The Kripke structure that corresponds to a collective of concurrent state machines can conceptually be simple to describe. Any valuation of the variables of all the FSMs, including its program counters, is a Kripke state. If the computation can possibly move from one Kripke state to another we have a link in the Kripke structure.

The generation of a Kripke structure for one sequential machine already causes concurrency issues due to the effect of the environment on the *external variables* (which the FSM reads, but the environment updates). External variables require some attention because we need to specify the level of temporal granularity at the border between the system and the environment. Inconsistencies must not be possible due to the environment changing a variable so fast that multiple evaluations within a ringlet give different results. By taking a snapshot of external variables at pre-specified points in time as described below, we effectively create sampling points that specify a deterministic boundary of interaction with the environment. Once this boundary is specified, the Kripke structure for one machine can be completed [2].

Our modelling tool enables three types of variables: variables local to one FSM, variables shared between the FSMs in the sequence, and external variables, whose value can be updated or modified by the environment. The semantics of the execution of a FSM is that the machine reads the values of all the external variables just before the commencement of evaluating the first of the expressions of the transitions leading out of a state (if there is an **OnEntry** part, after that part has completed). That is, a machine takes a snapshot of externally visible (and, more importantly: modifiable) variables only once per execution of a ringlet. Consequently, the value of any variable will remain consistent throughout the sequence of expression evaluations, without the external environment being able to change the valuation of these variables. However, when control leaves a FSM and passes to the next in sequence, the next machine may find external variables modified by a ringlet or the environment.

Therefore, we have several machines running concurrently, but they do so in a predefined schedule that can be mapped to a sequential program. This sequential program is modelled by a Kripke structure as is described in the model-checking literature [6, Chapter 2]. We have fully implemented this translation.

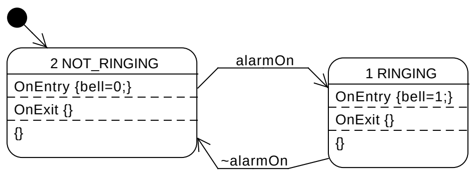
III. Case studies

A. Mine Pump

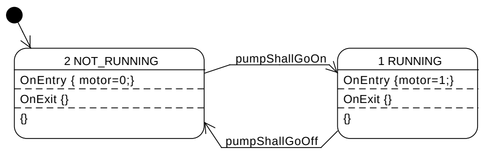
The mining pump is a case widely discussed in the literature [7]–[10] where software is controlling a safety-critical system. Our presentation follows Burns and Lister [11] as significant formality is provided. We use this case study because it enables the presentation of our methodology, exemplifies our methodology for constructing the model, for

Table I: Mine Shaft Pump Requirements.

Req.	Description
R1	The pump extracts water from a mine shaft. When the water volume has been reduced below the low-water sensor, the pump is switched off. When the water raises above the high-water sensor it shall switch on.
R2	An human operator can switch the pump on and off provided the water level is between the high-water sensor and the low-water sensor.
R3	Another button accessed by a supervisor can switch the pump on and off independently of the water level.
R4	The pump will not turn on if the methane sensor detects a high reading.
R5	There are two other sensors, a carbon monoxide sensor and an air-flow sensor, and if carbon monoxide is high or air-flow is low, and alarm rings to indicate evacuation of the shaft.



(a) States for the alarm.



(b) States for the pump.

Figure 1: Two FSMs control the mine pump.

```
%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 2: DPL coding of the conditions for switching the alarm to the state RINGING.

performing model checking, and for directly executing the models on a platform (thus fulfilling the promise of model-driven engineering). The requirements in natural language appear in Table I. The pump is in one of two states, *running* or *not running*, while simultaneously the alarm is in a *ringing* or *not ringing* state (see Fig 1). We label the transition from NOT_RINGING to RINGING by the predicate alarmOn which we consider a request to a hypothetical expert to indicate to us whether the alarm shall switch state from not ringing to ringing. The output section in Fig. 2 indicates such an expert makes judgements on alarmOn. The expert makes its judgement based on information about whether the CO_2-sensor indicates a high level or not, and whether the air-flow sensor indicates low flow or not. In DPL, having information on all of these inputs is not necessary. We state these desirable inputs in the input section of the DPL code in Fig. 2. By default, the alarm is not on; this is Rule A0. Rule A1 indicates that usually, a CO2SensorHigh causes the alarm to go on, and Rule A1 takes precedence over A0. Similarly, Rule A2 says that airFlowLow usually results in the alarm on, and Rule A2 takes precedence over A0. Compiling this rule system in the file Alarm.d using the +c option, we obtain an equivalent C expression for when the alarm shall ring.

$$\texttt{alarmOn} \equiv \texttt{CO2SensorHigh} \,||\, \texttt{airFlowLow}.$$

The result is the FSM in Fig. 3a.

Applying our techniques to the pump engine reveals several issues with the specification. First, although common sense suggest this, one must inspect the pre-conditions and post-conditions [11], to confirm that the conditions that turn the pump on are not the negation of those that turn it off. In particular, the pump goes on when the water level is high, but it remains on when the level drops below high (Requirement R1). Also, the pump must be running and the water level drop below the low sensor for it to stop. But it does not re-start as soon as the water level is above the below sensor. Unfortunately, Requirement R3 is seriously more ambiguous and the language used suggests that the operator's interface as well as the supervisor's interface for controlling the pump are both switches "of the same class" [11] (an assumption also shared by the Behavior Tree approach [9], [10]). We argue this first interpretation is inconsistent with the requirements and that our approach reveals this. Such switches are either on or off; holding only two exclusive states.

Under this first interpretation, we label the transition from NOT_RUNNING to RUNNING by the predicate pumpShallGoOn which we consider a request to an expert to indicate to us whether the pump shall be running on or not. The expert makes its judgement based on information about whether the low-water sensor is on or not, the high-water sensor is on or not, the operator button is on or not, whether the supervisor's button is on or not and whether the methane sensor has a higher reading or not. Again, in DPL, having information on all of these inputs is not necessary.

The question now becomes, when the pump should move from on to off. If we have no information, it shall remain on and not move (Rule N0). But if the low-water sensor in on, then the pump switches off (this is Rule N1, which takes over Rule N0). Rule N2 says that under water level above the low-water sensor and below the high-water sensor the operator can turn the pump off (N2 takes precedence over N0). And the supervisor can turn the pump off, by Rule N3 which takes over Rule N0. A high methane reading takes precedence and turns the pump off as well (Rule N4).

So, what should the response be concerning pumpShallGoOn? By default, the pump shall not go on. That is, the expert will be asked this question when the pump is off, and in the absence of no information, it should not recommend a change of state (this is Rule `P0`). Rule `P1` indicates that if the high-water sensor is on, then the pump goes on and this takes precedence over Rule `P0`. Rule `P2` indicates that the operator can turn the pump on if the water is between levels. Thus, `P2` overrides `P0`. However, with `P3` and `P4` the supervisor can turn the pump on and off. Nevertheless, all these previous conditions are ruled out if methane is high (Rule `P5`). The reader may have noticed that we have stated Rule `P3` in contradiction with Requirement `R3`. This is because if we add the precedence `P3>P0` and write

$$\texttt{P3:supervisorButtonOn=>pumpShallGoOn.}$$

enabling the switch of the supervisor to overrule the low-water sensor, then (by compiling these rules) we obtain

$$\texttt{pumpShallGoOff} \equiv \texttt{!supervisorVuttonOn||methaneSensorHigh}.$$

That is, with this interpretation, *the water levels are irrelevant for deciding if the pump shall be off* (and therefore the operator is redundant as well)! All depends on the methane level, which overrules everything else to switch the pump on or off. Equally, the supervisor switch to turn the pump on and off "independently of water levels" overrules these, rendering the corresponding requirements irrelevant. Similarly, if one would like the supervisor's button to overrule the button of the operator, the operator's button is forced to agree with the supervisor's switch, as the supervisor's switch would always take precedence.

Therefore, one needs to be prepared to accept that the low-water sensor takes precedence over the supervisor's switch, which makes sense if running the pump without water is dangerous. A fact that is verified with model checking (or with simulation) in our methodology.

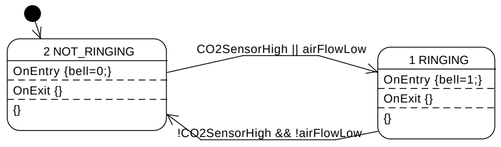
Compiling this rule system using the `+c` option, we obtain equivalent `C` expressions for when the pump shall be on and for when it shall go off. Namely,

$$\begin{aligned}\texttt{pumpShallGoOff} \equiv\ & \texttt{lowWaterSensorOn}\ \ \texttt{||!supervisorButtonOn}\\ & \texttt{||}\ \ \texttt{methaneSensorHigh}\\ & \texttt{||}\ \ (\texttt{!highWaterSensorOn \&\& !operatorButtonOn})\end{aligned}$$

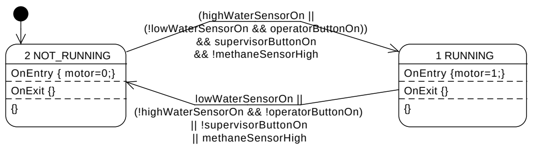
and
$$\begin{aligned}\texttt{pumpShallGoOn} \equiv\ & \texttt{!methaneSensorHigh\&\&supervisorButtonOn}\\ & \texttt{\&\&}(\texttt{highWaterSensorOn}\\ & \texttt{||}\{\texttt{!lowWaterSensorOn \&\& operatorButtonOn}\}).\end{aligned}$$

Notice the asymmetry between the supervisor and operator conditions for pump start! Combining this with the FSM for the alarm results in the model in Fig. 3.

Nevertheless, we suggest that there is a second interpretation, i.e., the supervisor's interface is a three state control, that has an inactive state. In the inactive state, it does not overrule any of the other conditions that determine the running of the pump. But the supervisor can activate the switch to on, or off, switching the pump on and off regardless of water levels. That model for supervisor control is presented



(a) Boolean expression for alarm state transitions.



(b) Boolean expressions for pump state transitions.

Figure 3: Two FSMs controlling the mine pump with two state-switches for the operator and the supervisor.

in Fig. 4 and the new logic theory appears in Fig. 5. We now compile this logic theory into the corresponding equivalent expressions, namely,

$$\begin{aligned}\texttt{pumpShallGoOn} \equiv\ & \texttt{!methaneSensorHigh}\ \ \texttt{\&\& !indicateOff}\\ & \texttt{\&\&}\ (\texttt{indicateOn}\\ & \quad \texttt{||}\ \{\texttt{!lowWaterSensorOn}\\ & \quad\quad \texttt{\&\&}\ [\texttt{highWaterSensorOn}\\ & \quad\quad\quad \texttt{||}\ \ \texttt{operatorButtonOn}]\}).\\ \texttt{pumpShallGoOff} \equiv\ & \texttt{methaneSensorHigh}\ \ \texttt{||}\ \ \texttt{indicateOff}\\ & \texttt{||}\ \ (\texttt{!}\ \ \texttt{indicateOn}\\ & \quad \texttt{\&\&}\ \ \{\texttt{lowWaterSensorOn||}\\ & \quad\quad (\texttt{!highWaterSensorOn}\\ & \quad\quad\quad \texttt{\&\& !}\ \ \texttt{operatorButtonOn})\}.\end{aligned}$$

Replacing the transition labels in Fig. 1b, we obtain our model for the second interpretation of this case study.

The literature offers details of formal verification for only three properties [9].

Property-1 *"If the CO_2 is high, the alarm to evacuate personnel must ring."*

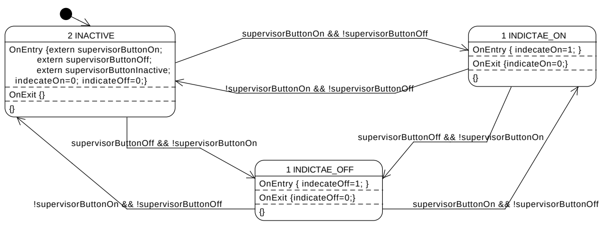
Property-2 *"If the airflow is low, the alarm to evacuate personnel must ring."*

Property-3 *"If the methane level is high, the pump must be turned off."*

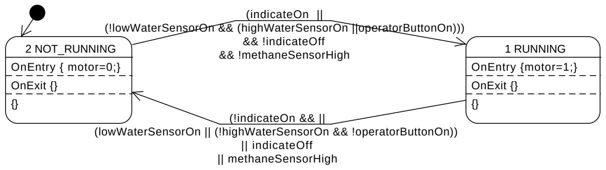
Fig. 6 shows our coding if these properties in Linear-Time Temporal Logic (LTL) [6] but using the syntax for `NuSMV` and suitable for our derived Kripke structure. Similarly to the LTL coding used earlier [9], the are a few internal Kripke states, before the system rings the bell. Thus, for example, for the first property, its LTL describes that, if the airflow is low for 5 consecutive Kripke states, then the bell rings.

Here we go beyond previously analysed properties for this case study and explore the following.

Property-4 *"If the supervisor turns the pump off when running, the pump will be turned off."*


2 INACTIVE
OnEntry {extern supervisorButtonOn;
extern supervisorButtonOff;
extern supervisorButtonInactive;
indecateOn=0; indicateOff=0;}
OnExit {}
{}
supervisorButtonOn && !supervisorButtonOff
1 INDICTAE_ON
OnEntry { indecateOn=1; }
OnExit {indicateOn=0;}
{}
!supervisorButtonOn && !supervisorButtonOff
supervisorButtonOff && !supervisorButtonOn
supervisorButtonOff && !supervisorButtonOn
1 INDICTAE_OFF
OnEntry { indecateOff=1; }
OnExit {indicateOff=0;}
{}
!supervisorButtonOn && !supervisorButtonOff
supervisorButtonOn && !supervisorButtonOff

(a) Model for the three-state supervisor control.



(b) The model for the pump's software control that receives signals from a three-state supervisor control.

Figure 4: The three-state supervisor pump model consists of these two FSMs and the FSM in Fig. 3a 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,~operatorButtonOn}
    => pumpShallGoOff.
        N2>N0.
N3: indicateOff => pumpShallGoOff.
        N3>N0.
N4: methaneSensorHigh => pumpShallGoOff.
        N4>N0.

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

Figure 5: DPL coding of the conditions for switching to the state `RUNNING` using a three-state supervisor control.

```
LTLSPEC
G  (E$$CO2SensorHigh = 1  -> X (  E$$CO2SensorHigh = 0  | bell = 1 |
                               X ( E$$CO2SensorHigh = 0 |  bell =1 |
                                        X(bell=1 | X(bell=1) ) )))
LTLSPEC
G  (E$$airFlowLow = 1  -> X (  E$$airFlowLow = 0  | bell = 1 |
                               X ( E$$airFlowLow = 0 |  bell =1 |
                                       X(bell=1 | X(bell=1) ) )))
LTLSPEC
G  (E$$methaneSensorHigh = 1  -> X ( motor = 0 | X (  motor =0 )))
```

Figure 6: `NuSMV` coding of the first three properties for the Mine Pump.

Property-5 *"If the operator turns its 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 comes on when the water is above the high water sensor (and the low-water sensor's signal is consistent with this), unless the supervisor turn it off or there is high methane."*

These properties hold both for the models in Figs 3 and 4. However, the reader may notice that we do not need to verify the properties of the alarm at the same time as those that turn the pump on and off, as these are independent. Finding there are no dependencies facilitates model checking as the state space of the separate Kripke structure is much smaller [12]. Importantly, this proves that for these case studies, our approach enables full model checking.

The difference between the model in Fig. 3 and the model in Fig. 4 are the following properties.

Property-7 *"If the supervisor sets the switch as 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 comes off."*

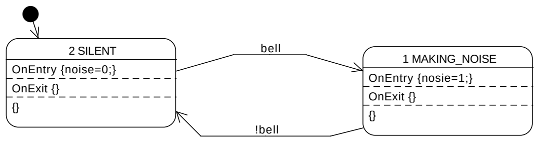
Property-8 *"If there is low methane, low water, and the pump is not running, but the supervisor puts the switch to on, then the pump comes on."*

Property-9 *"If 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 comes off (independently of the supervisor's switch)."*

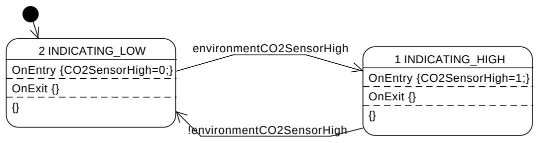
Property 9 holds for the model in Fig. 3 but not Fig. 4. However, Properties 7 and 8 hold for the model in Fig. 4.

To validate our models, we have enabled our interpreter to control a Lego NXT and built a model of the pump with touch-sensors and floating balls that, when the water level rises high enough, will trigger the sensor. Unfortunately the NXT only has four ports for sensors, thus we do not observe the behaviour of the alarm. Also, the three-state supervisor's sensor is simulated by cycling trough indicating on, indicating off, and inactive for each push and release of the corresponding touch sensor. Although this is a rather improvised artefact with unreliable sensors, we have a video that shows that the behaviour of the pump under the control of the model in Fig. 4 is correct. We trust the correctness derived from the model checking as the utmost proof. The simulation or emulation merely illustrates the point of direct execution of the model on a platform. A video of the pump executing the model appears on `http://youtu.be/y4muLP0jA8U`.

Our ability to formally verify the models permits to analyse the impact and severity of potential failures. We follow the earlier *fault injection* approach [9]. We then build the FMEA table of failures and consequences. This is illustrated



(a) Model of the bell in the alarm.



(b) Model of the CO_2-sensor in the alarm.

Figure 7: Two FSMs that model an actuator and a sensor, respectively, in the mine pump.

with the interpretation of a 3-state switch for the supervisor (the process is analogous for the alternative interpretation). Adding a sensor or actuator to the model in Fig. 4 means explicitly modelling these devices (while Fig. 4 models the controlling software). Thus, we now also have FSMs for the devices; one sensor (the CO_2-sensor) and one actuator (the bell) are shown in Fig. 7. These correspond to the alarm controller in Fig. 3a. If these machines are correct (no failures), the generation of the Kripke structure (with more internal transitions, as now it models relaying from the controller to the actuator and execution of **OnEntry**, **OnExit**, and internal sections of each) and the validation of all properties succeeds. However, the interesting part is when we construct these models with a defect. For example, if the CO_2-sensor fails to detect a high level of CO_2 in the environment. We inject this failure by modifying the **OnEntry** section of the state `INDICATING_HIGH` from `CO2SensorHigh=1` to `CO2SensorHigh=0`. We re-build the Kripke structure and perform the model checking, which shows now that Property 1 fails. Systematically testing the failures that a sensor may stay stuck in high or low, or an actuator stays stuck off or on, we obtain all rows of the FMEA table corresponding to level one failures (Table II). Moreover, we obtain more possible failures than the previous analysis with Behavior Trees [9]. We also analyse the case a sensor is inactive. For example, the CO_2 sensor stuck on high corresponds to the **OnEntry** section of the `INDICATING_LOW` in Fig. 7b having the statement `CO2SensorHigh=1`, instead of the correct `CO2SensorHigh=0`. However, another type of failure is that the statement is actually removed and the **OnEntry** section of the state had no action. In this example, the consequences are not much dissimilar to the stuck value, but in other case studies, the consequences are different. The fact that Property 1 does not fail when the CO_2-sensor is stuck high, is because being on high is what the sensor should do when there is CO_2 in the environment and cause the alarm to ring. Obviously, there are other properties than the 8 we have listed to complete verification of the model. However, we are here already showing far more properties than those verified earlier in the literature.

Table II: FMEA table at level one for the model in Fig. 4.

Failures	Consequences							
	Property that fails							
	1	2	3	4	5	6	7	8
CO_2-sensor stuck high								
CO_2-sensor stuck low	×							
Airflow sensor stuck high		×						
Airflow sensor stuck low								
Bell stuck ringing								
Bell stuck not ringing	×	×						
Supervisor button stuck in on				×			×	
Supervisor button stuck in off						×	×	×
Operator button stuck in on					×			
Operator button stuck in off							×	
Methane sensor stuck in high						×		×
Methane sensor stuck in low			×					
(High water) sensor stuck in on					×		×	
(High water) sensor stuck in off						×	×	
(Low water) sensor stuck in on						×		
(Low water) sensor stuck in off					×	×		×
Motor stuck running			×	×	×		×	
Motor stuck not running						×		×

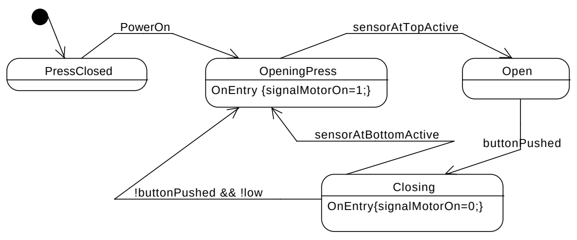
Table III: Industrial Metal Press Requirements.

Req.	Description
R1	The plunger is initially resting at the bottom with the motor off.
R2	When power is supplied, the controller shall turn the motor on, causing the plunger to rise.
R3	When at the top, the plunger shall be held there until the operator pushes and holds down the button. This shall cause the controller to turn the motor off and the plunger will begin to fall.
R4	If the operator releases the button while the plunger is falling slowly (above PONR), the controller shall turn the motor on again, causing the plunger to start rising again, without reaching the bottom.
R5	If the plunger is falling fast (below PONR) then the controller shall leave the motor off until the plunger reaches the bottom.
R6	When the plunger is at the bottom the controller shall turn the motor on: the plunger will rise again.

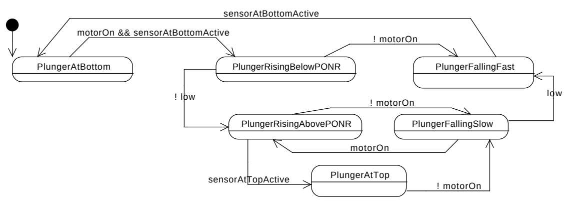
B. Industrial Metal Press

The industrial metal press [13] has been studied in the literature of model checking for failure analysis [9], [14] and full details appear in the associated report [15]. Table III reproduces the requirements [15]. Once again, later papers [9], [15] that cite the original source [13] capture requirements differently or incompletely. One important contrast is the requirement that once the plunger has come down, it shall stay down until the "human operator releases the button and inserts and removes sheets" [14]. In the original sources [13] there is even an additional infrared-line sensor; once the plunger is down, the button must be released by the operator and the line cut for the plunger to move up again.

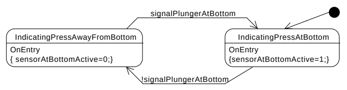
Because the On/Off button is not modelled correctly, and the infrared-link is not modelled at all, modelling as described in the Design Behavior Tree (DBT) [15, Fig. 4 and HighResolution gif] results in a pathological cyclical behaviour of the system. In particular, while the human operator keeps the button pushed (and the power is on), the plunger rises to the top (with the motor on), reaches the top (the motor goes off) falls down, and rises up again in a cycle that is contrary to the original requirements and what a human would interpret as safe.



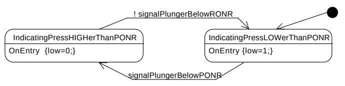
(a) States of the Bottom Sensor.



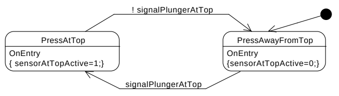
(b) States for the Plunger.


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

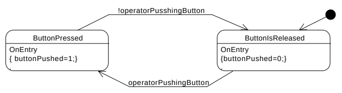
(c) States of the Bottom Sensor.



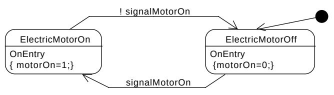
(d) States of the PONR Sensor.



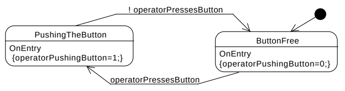
(e) States of the Top Sensor.



(f) States of the Button.



(g) States of the Electric Motor.



(h) States of the Operator.

Figure 8: Complete model of the industrial press that mimics the Design Behavior Tree [15, Fig. 4 and high-res gif].

First, we have constructed a model that reproduces the DBT [15, Fig. 4] to focus on the model checking aspects. This model that mimics such a DBT appears in Fig. 8.

The following safety conditions, postulated in LTL, were proposed for formal verification [9], [15].

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 modelled by the plunger falling fast, then the motor should remain off."*

Property-3 *"If the plunger is falling above the PONR, a state modelled by falling slow, and the operator releases the button, the motor should eventually turn on, before the plunger changes state."*

Property-4 *"The motor should never turn off while the plunger is rising"*.

We will not elaborate here on our formal verification of all these properties with NuSMV, as the process is similar to the previous case study presented in this paper. We rather emphasise that these properties are not sufficient to completely validate the model. However, the analysis using our approach does suggest a correction. Fig. 9 depicts this solution that enables to remove the anomaly by waiting for a signal that turns the system on again before rising again. This again shows that our approach compares favorably with Behavior Trees. To further illustrate this point, we have deployed these models for execution in two platforms, an Aldebaran NAO robot and a Lego NXT. A video of the system emulating the behaviour as formalised from the Behavior Tree approach and its corrections is available at `http://youtu.be/blUpMdH14pM`.

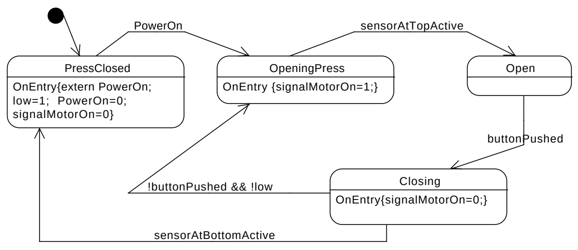


Figure 9: Corrected model from Fig. 8a (based on DBT modelling from the literature). This model correctly pauses for a signal that turns the system on before rising again.

IV. Conclusion

We have shown that models built using multiple finite-state machines (whose transitions are labelled by expressions in a common-sense logic) can be formally verified using standard model-checking technology. Such formal verification can be further validated, as, with our approach, these models can be simulated or directly executed on multiple platforms. While formal verification via model-checking provides complete confidence in the model itself, validation, through simulation or execution of whether a requirement (a property of its behaviour) is satisfied, is important as a human language system specification on its own cannot formally be verified for completeness.

We have also shown that our approach can be used to systematically derive FMEA tables, through the production of efficient, manageable Kripke structures corresponding to component failure. This contrasts with the Behavior Tree approach, where several threads and channels of communication occur within the Behavior Tree, resulting in computations too complex to perform any sort of model-checking on, without a large number of simplifying assumptions (even then, CPU times that in some cases exceed four days for verification [9] have been reported). None of our verifications required more than a few seconds, although in some cases, the generated Kripke structures result in files of several megabytes.

Of more concern is the fact that previous research of this type has assumed that updates of internal variables always takes precedence over external events (it is argued 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 [9] and their model checking is not sound). We do not 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, we do not make any assumptions about the speed with which any sensor is updated. In our models, sensors can also be updated much faster than the software may be able to execute (e.g. when delayed in a multi-tasking operating system). This is of particular interest in systems such as the Industrial Press where the position of the press below or above the PONR can cause very fast updates.

Thus, the sequential execution of the concurrent finite-state machines that represent component behaviour of a larger system, models the behaviour of the system in an effective and efficient way for model-checking and simulation. We believe that Behavior Trees and our technique are complementary. Behavior trees have the advantage that models can always be drawn without crossings, making them appealing to human designers. They also are supported by methodologies that derive them directly from the linear story telling of requirements. Therefore, Behavior Trees could be the starting model (as in the case illustrated by the Industrial Press), from which finite-state machines could be derived.

Acknowledgements

We thank our colleagues from the Mi-PAL team as well as the Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS), at the Universidad Nacional Autónoma de México (UNAM).

References

[1] D. Schmidt, “Model-driven engineering,” *IEEE Computer*, vol. 39, no. 2, 2006.

[2] V. Estivill-Castro and D. A. Rosenblueth, “Model-checking of transition-labeled finite-state machines,” in *Proc. 2011 Int. Conf. Adv. Softw. Eng. & its Applications*, ser. Comm. in Computers and Inf. Sc., T.-H. e. a. Kim, Ed., vol. 257. Jeju Island, Korea: Springer Verlag, December, 2011, p. 61.

[3] D. Billington and A. Rock, “Propositional plausible logic: Introduction and implementation,” *Studia Logica*, vol. 67, pp. 243–269, 2001, iSSN 1572-8730.

[4] 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.

[5] T. Merz, P. Rudol, and M. Wzorek, “Control system framework for autonomous robots based on extended state machines,” in *Proceedings of the International Conference on Autonomic and Autonomous Systems, ICAS ’06*, Silicon Valley, CA, July 16-18 2006, p. 14.

[6] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 2001.

[7] S. 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.

[8] M. Sloman and J. Kramer, *Distributed systems and computer networks*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1987.

[9] 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.

[10] K. Winter and N. Yatapanage, “The mine pump case study,” University of Queensland, Tech. Rep., supplement in `www.itee.uq.edu.au/~docs/FMEA`.

[11] A. Burns and A. Lister, “A framework for building dependable systems,” *The Computer Journal*, vol. 34, no. 2, pp. 173–181, 1991.

[12] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. Holzmann, “Validating requirements for fault tolerant systems using model checking,” in *3rd International Conference on Requirements Engineering (ICRE ’98), Putting Requirements Engineering to Practice*. Colorado Springs, CO, USA: IEEE Computer Society, April 6-10 1998, pp. 4–13.

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

[14] 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.

[15] K. Winter and N. Yatapanage, “The metal press case study,” University of Queensland, Tech. Rep., supplement in `www.itee.uq.edu.au/~docs/FMEA`.