Module Isolation for Efficient Model Checking *and its Application to FMEA in Model-Driven Engineering*

Vladimir Estivill-Castro and René Hexel
Griffith University, Brisbane, Australia

Outline

- Motivation
  - Model-Driven Development (MDD)
  - Finite State Machines
- Logic-labeled Finite-State Machines
- Formal Verification and Model Checking
- Use for Failure Mode Effect Analysis (FMEA)
- Examples and Results
- Conclusions

Model-Driven Development

- Widely successful approach to developing software
- Ensures traceability, validation against requirements, and platform independence
- Tools and techniques are resulting in faster and simpler (easier to maintain) products and applications than traditional language parser/ compiler or interpreter approaches

Finite-State Machines

- Finite state machines (FSMs) are ubiquitous to describe system behavior
  - *QP* (Samek, 2008), *Bot- Studio* (Michel, 2004) *StateWORKS* (Wagner et al., 2006) and MathWorks *StateFlow*. The UML form of FSMs derives from OMT (Rumbaugh et al., 1991, Chapter 5), and the MDD initiatives of Executable UML (Mellor and Balcer, 2002).
- Large penetration in industrial settings
- Concurrent execution of FSMs and model checking faces the challenge that

$$\#\ \text{of states of the system} = \prod \#\ \text{of states of each subsystem}$$

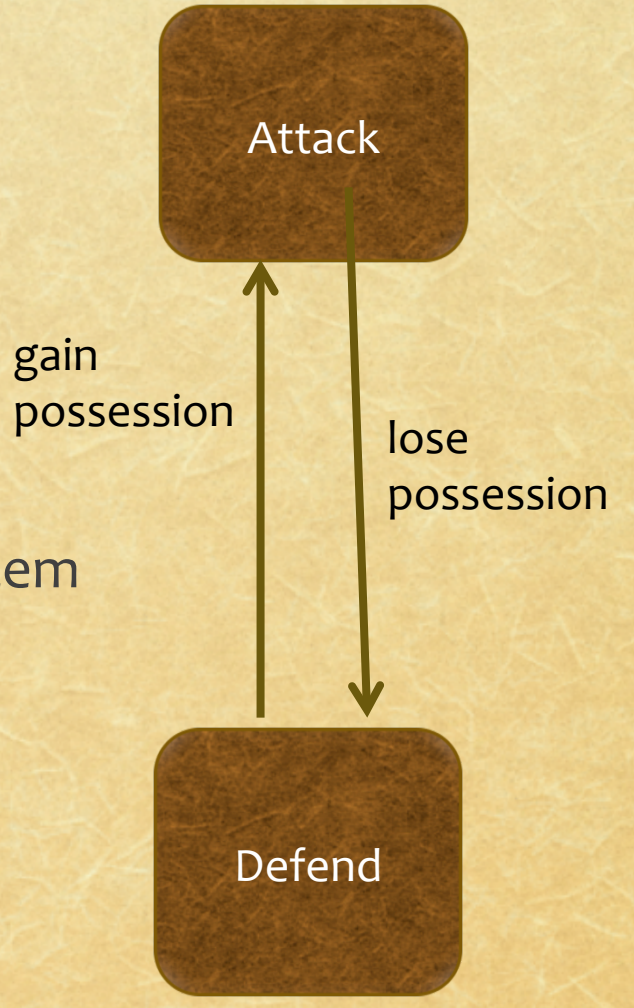
Formal verification and model checking

- Formal verification validates the system, but
  - Systems are also examined using fault injection and extensive Failure Mode Effects Analyses (FMEAs)
- The model-checking exercise is repeated in a system with an injected fault to determine the effect of such fault
- Complemented by simulation of other components and their faults
  - Increases the number of states in the system

Event-driven FSMs

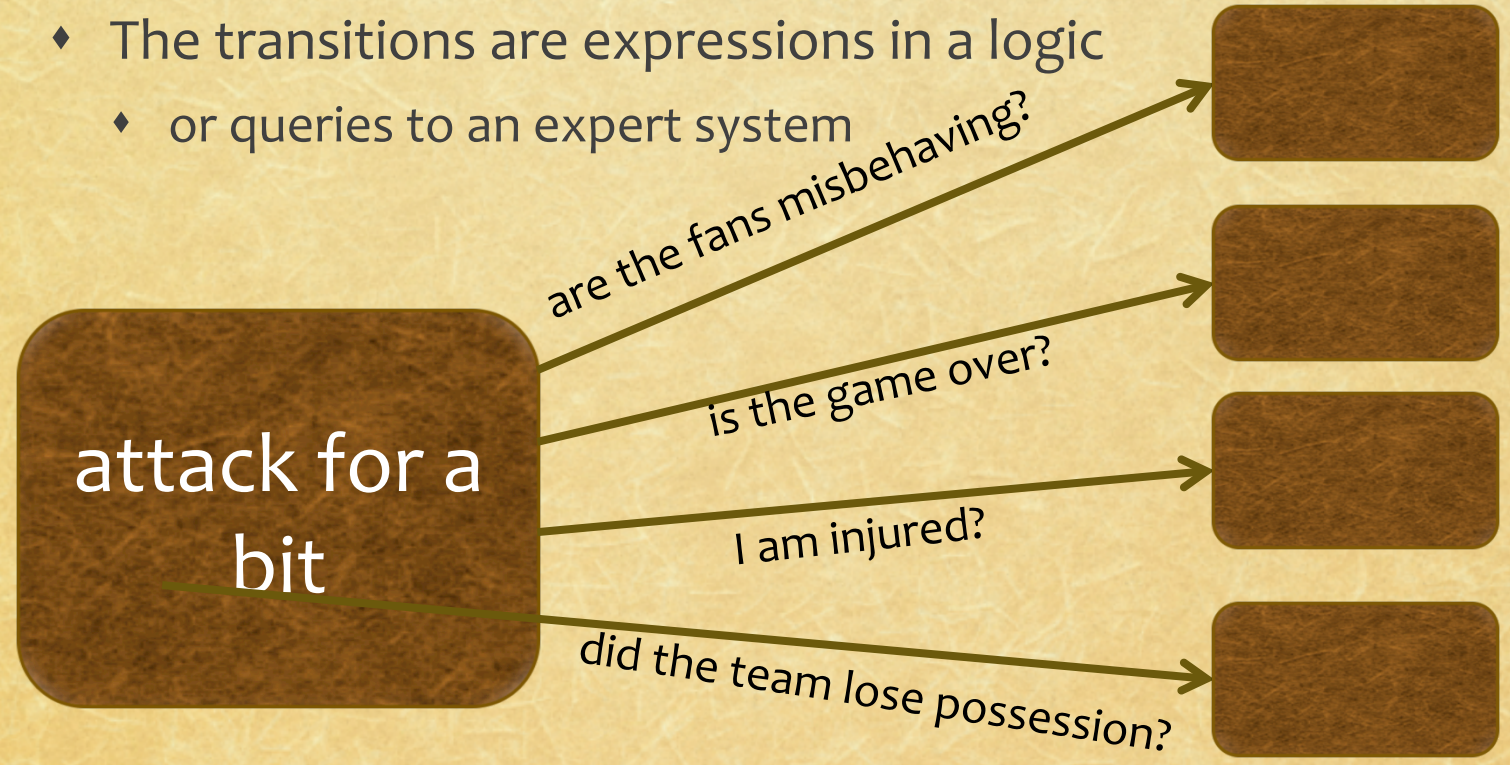
Most common approach

- System is in a state
  - waiting
  - does not change what is
    - doing/happening
  - until event arrives
- Events change the state of the system



Logic-labeled FSMs

- A second view of time (since Harel's seminal paper)
  - Machines are not waiting in the state for events
  - The machines drive, execute
  - The transitions are expressions in a logic
    - or queries to an expert system



Illustration

- The Micro-wave Oven
  - (ubiquitous in the literature of model-checking and model-driven development)

Requirements	Description
R1	There is a single control button available for the use of the oven. If the oven is closed and you push the button, the oven will start cooking (that is, energize the power-tube) for one minute
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3	Pushing the button when the door is open has no effect.
R4	Whenever the oven is cooking or the door is open, the light in the oven will be on.
R5	Opening the door stops the cooking. and stops the timer and does not clear the timer
R6	Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
R7	If the oven times out, the light and the power-tube are turned off and then a beeper emits a warning beep to indicate that the cooking has finished.

Complete model of the microwave oven

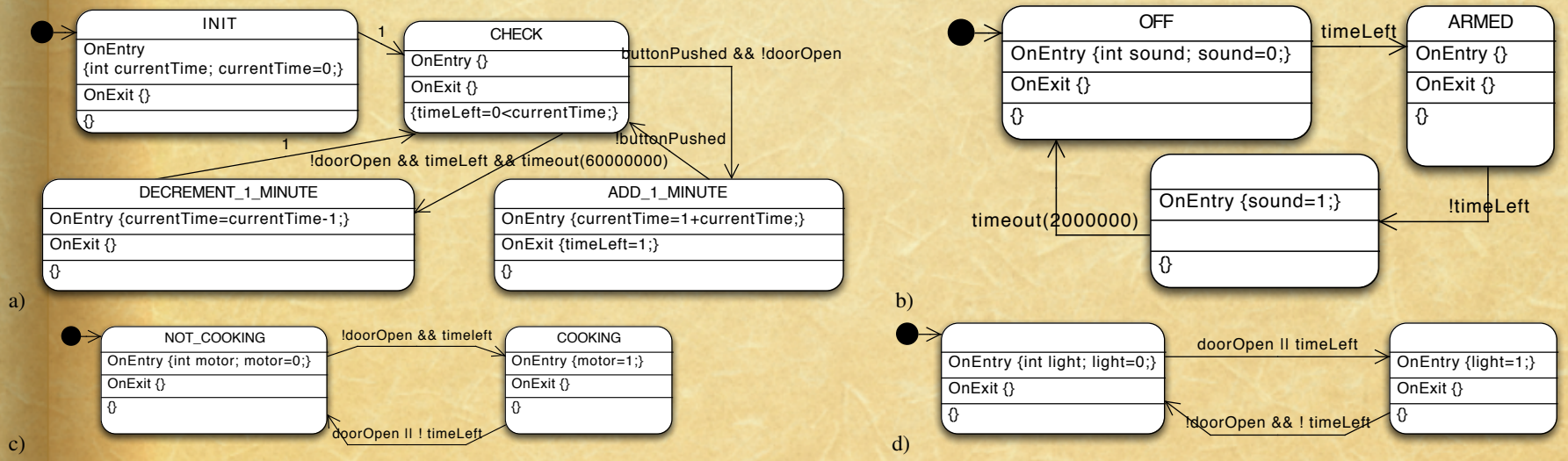


Figure 1: Complete model of one-minute microwave. a) A 4-state FSM for the timer. b) A 3-state machine for controlling the bell. c) A 2-state machine for controlling the cooking engine. d) A 2-state machine for the light.

Model Checking and Validation

- Properties
  - **Property 1:** *Necessarily, the oven stops (after* several *steps, i.e. a small, finite number of transitions in the Kripke structure) after the door opens."*
  - **Property-2:** "*It is necessary to pass through a state in which the door is closed to reach a state in which the motor is working and the machine has started."*
  - **Property-3:** "*Necessarily, the oven stops(after* several *steps, i.e. again, a small, finite number of transitions in the Kripke structure) after the timer has expired."*
  - **Property-4:** "*Cooking may go on for ever (e.g. if the user repeatedly keeps pressing the* add *button while the timer is still running)."*

Formal description of the Property in LTL

- Using NUSMV's code
  - *"the cooking must stop if the door is held open"*

```
SPEC
AG( (E$$doorOpen=1 & M0$$motor=1) ->
        AX(  (E$$doorOpen=1 -> M0$$motor=0) | AX(
                (E$$doorOpen=1 -> M0$$motor=0) | AX(
                (E$$doorOpen=1 -> M0$$motor=0) | AX(
                (E$$doorOpen=1 -> M0$$motor=0) | AX(
                (E$$doorOpen=1 -> M0$$motor=0) | AX(
                (E$$doorOpen=1 -> M0$$motor=0) | AX(
                (E$$doorOpen=1 -> M0$$motor=0) | AX(
                M0$$motor=0)))))))))
```

Failure Mode Analysis

- New components come into place

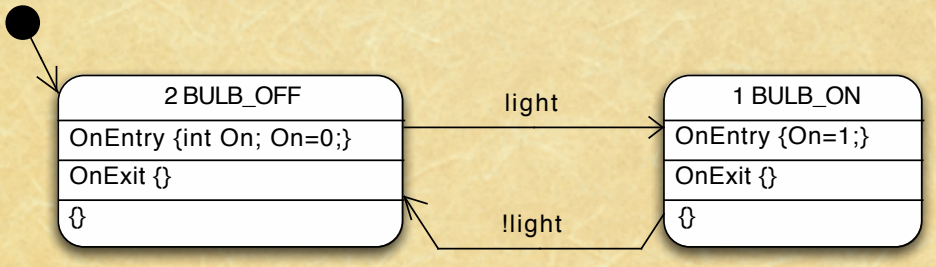


Figure 3: A model of the light bulb hardware component.

- Fault injection determines the effects
  1. to remove behavior from the model (an *omission failure*) and test all properties, and
  2. to modify (a *value failure*) behavior and test all properties.

Identification of independent sub-modules

- Whiteboard infrastructure holds the shared variables
- Sequential execution control the state explosion
- FSMs have a READ phase before they execute a ringlet
- An FSM's WRITE of the variables is not in a race condition with other FSMs
- The REQUIRES set of an FSMs is the set of shared variables it reads a value from
- The PROVIDES set of an FSM is the set of shared variables it modifies

We identify the (effect) dependencies among FSMs

- A dependency graph
- Nodes are the behavior modules (FSMs)
- a directed edge from FSM M_1 to FSM M_2 if the *REQUIRES* set of M_2 has a non-empty intersection with the *PROVIDES* set of M_1.

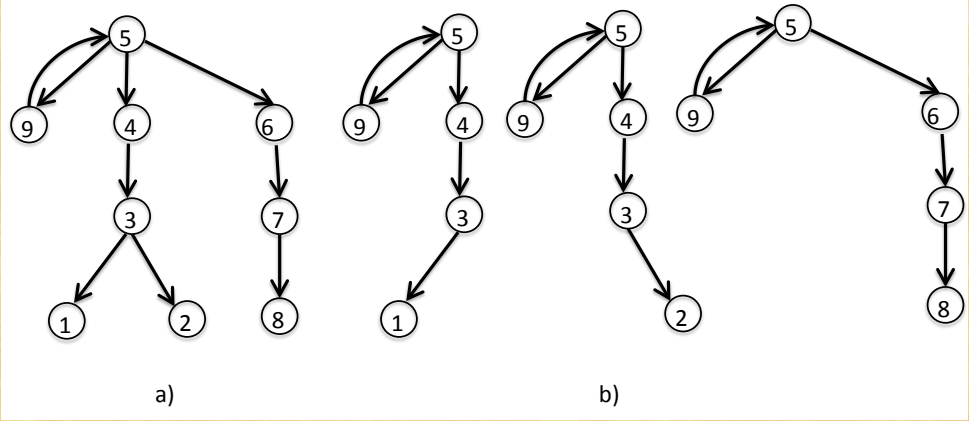


Figure 4: A dependency graph, and b) its cover into 3 components.

Observations

- **Definition:** If v_1 is a vertex of in-degree NOT 0, A_{v1} is the set of ancestor of v_1 and includes v_1
- **Lemma 3.1.** *For any vertex* $u_1 \in A_{v1}$ *, there is a directed path from* u_1 *to* v_1 *in G; and therefore the* WRITE *set of* u_1 *may influence the* READ *set of* v_1 *.*
- **Observation 3.2.** *If there is a directed path from a node* v_1 *to a node* v_2*, then* v_1 *and* v_2 *must be analyzed jointly.*

The cover

- A decomposition of the graph $G = (V, E)$ of dependencies into a cover $\mathbf{C} = \{C1,\ldots,C_t\}$ so that
  1. every node is included; that is $\bigcup_{C \in \mathbf{C}} C = V$, (is a cover)
  2. each component $C \in \mathbf{C}$ of this cover has the property that if *u* and *v* are vertices in C, then
     a. there is a path in C form *u* to *v* or
     b. a path in C from *v* to *u*,
  3. each component is ancestor-maximal, that is, there is no vertex *v* outside C so that there is a path from *v* to some vertex $u \in C$.

Algorithm

- Depth First Search
  - Classification of edges
    - ***tree edges*** which lead to new vertices during the search and form the topological-sort tree,
    - ***forward edges*** which go from ancestors to proper descendants but are no tree edges
    - ***back edges*** which go from descendants to ancestors, *cross edges* which go between vertices that are neither ancestors not descendants of one another.
- Find maximal ancestors of v_1 by Depth First Search in reverse direction
- Directed forward Depth First Search from Maximal ancestors

When do we find components

- there is a node *v* that has two or more children in the topological-sort tree,
- there is no back edge from any descendant of *v* in the topological-sort to an ancestor of *v*
- there is no forward edge from an ancestor of *v* to a descendant of *v*.
- Plus some other checks
  - see paper

Example with the microwave oven

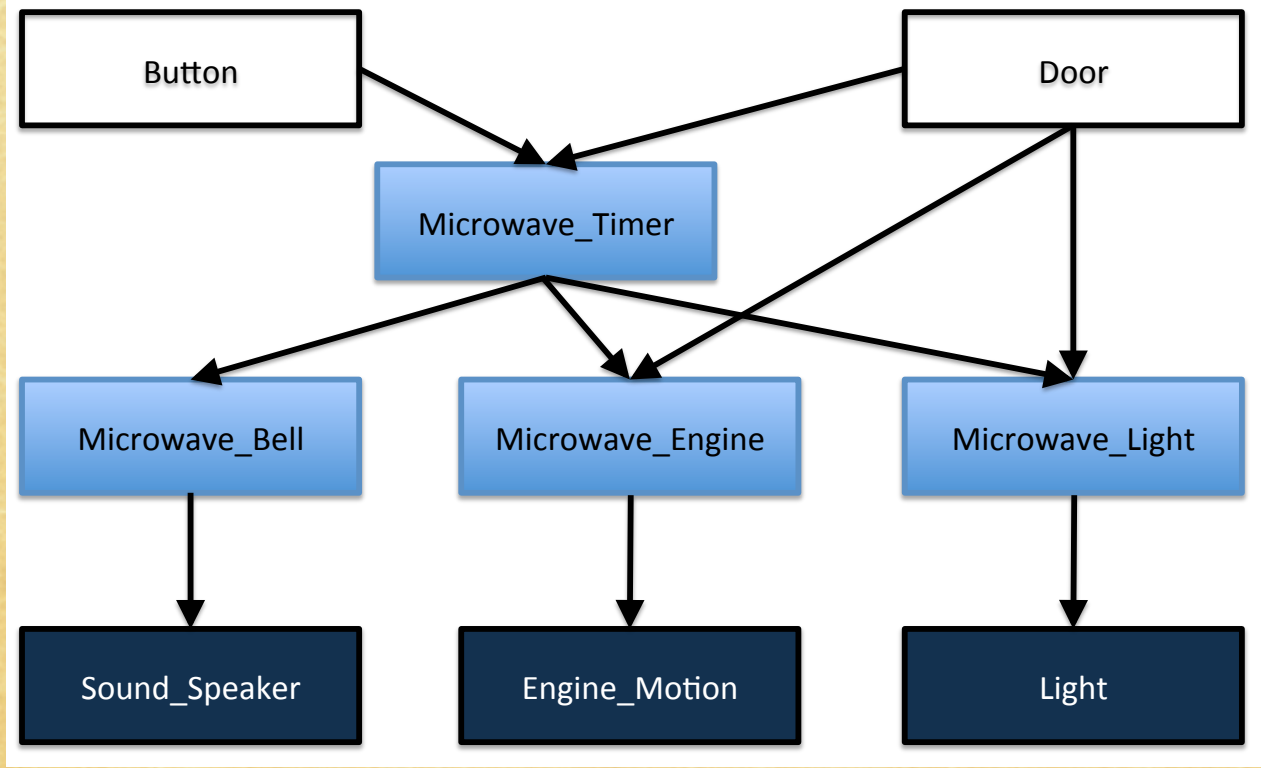


Figure 5: The dependencies of the modules of the `Microwave_Oven`.

The cover

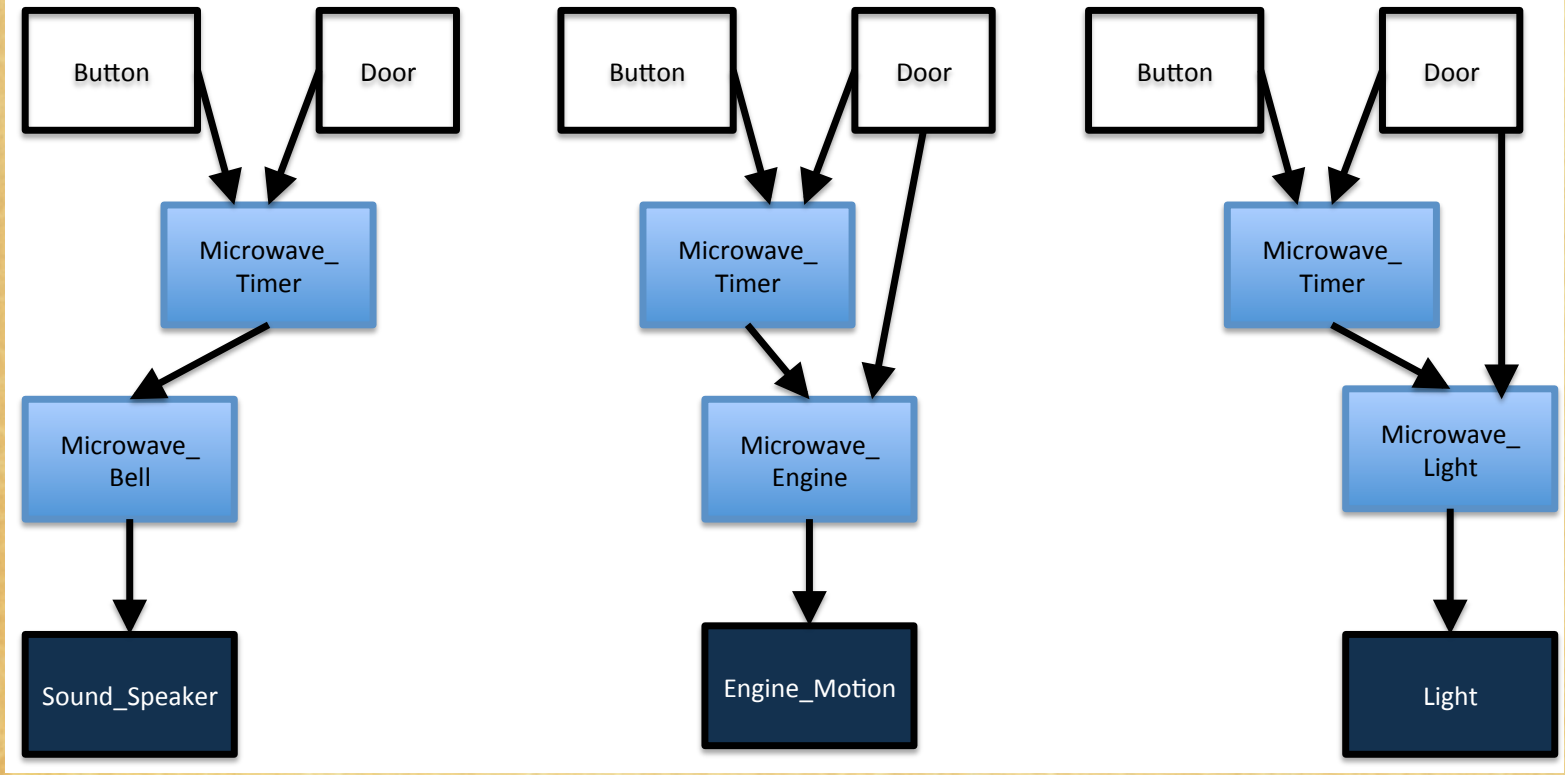


Figure 6: The cover of the dependencies graph (Fig. 5) into ancestor closed and maximal components.

Comparison

Table 2: Comparisons Kripke structure size (`NuSMV` file size) and generation time (`gufsm` CPU time) of the `Microwave_Oven` case study.

Component	CPU Time	Space
Combined graph	2,557.32 s	287,877,511 bytes
Bell subgraph	0.27 s	2,817,073 bytes
Engine subgraph	0.22 s	2,457,880 bytes
Light subgraph	0.22 s	2,458,762 bytes

Another example, the mine pump

Table 3: `Mine_Pump` requirements.

Req.	Description
R 1	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.
R 2	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.
R 3	Another button accessed by a supervisor can switch the pump on and off independently of the water level.
R 4	The pump will not turn on if the methane sensor detects a high reading.
R 5	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.

Logic-labeled FSMs for the mine pump

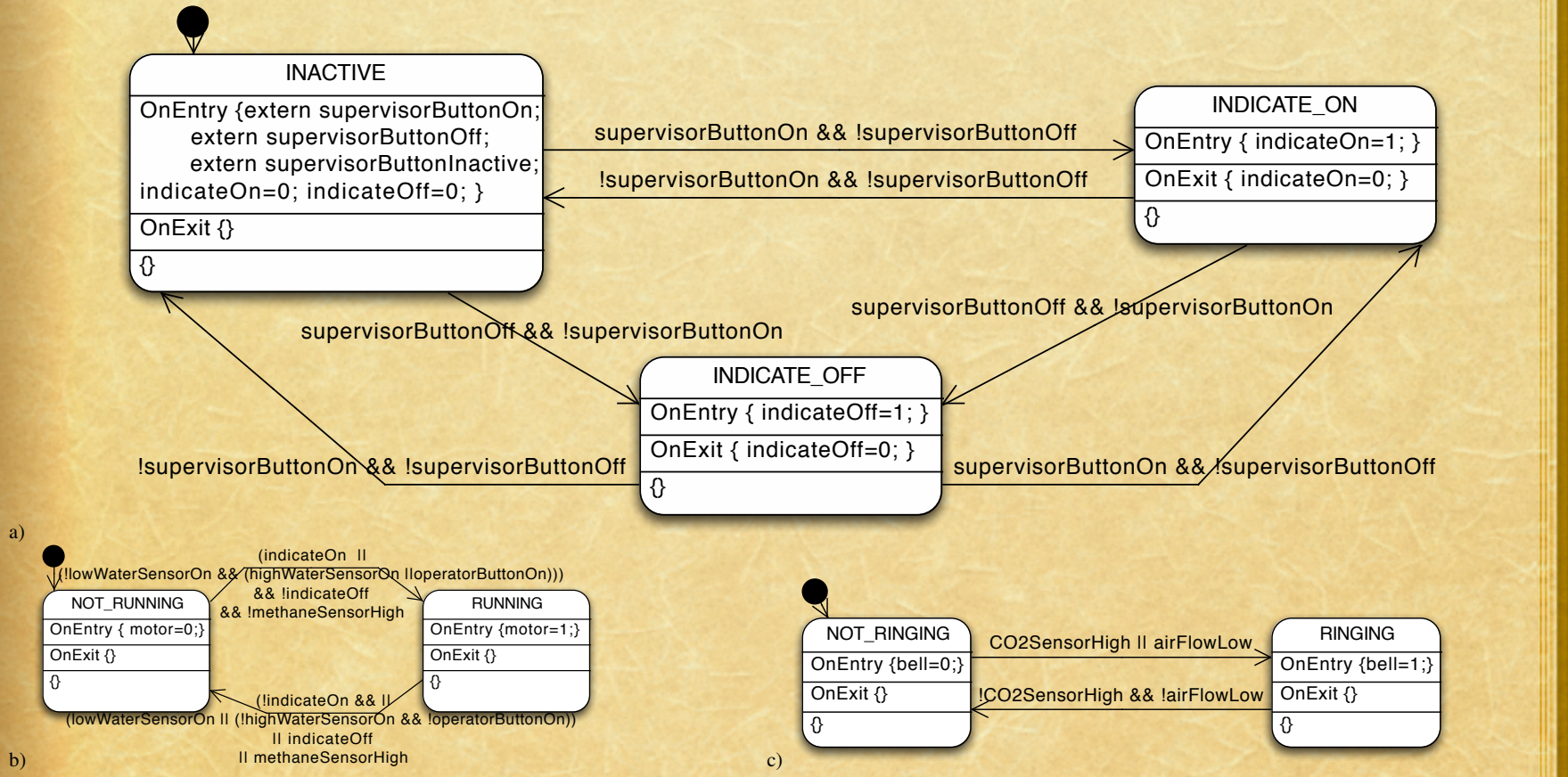


Figure 7: Complete model of the mine pump. a) A 3-state FSM for the supervisor. b) A 2-state machine for controlling the pump. c) A 2-state machine for controlling the alarm.

The graph for the mine pump

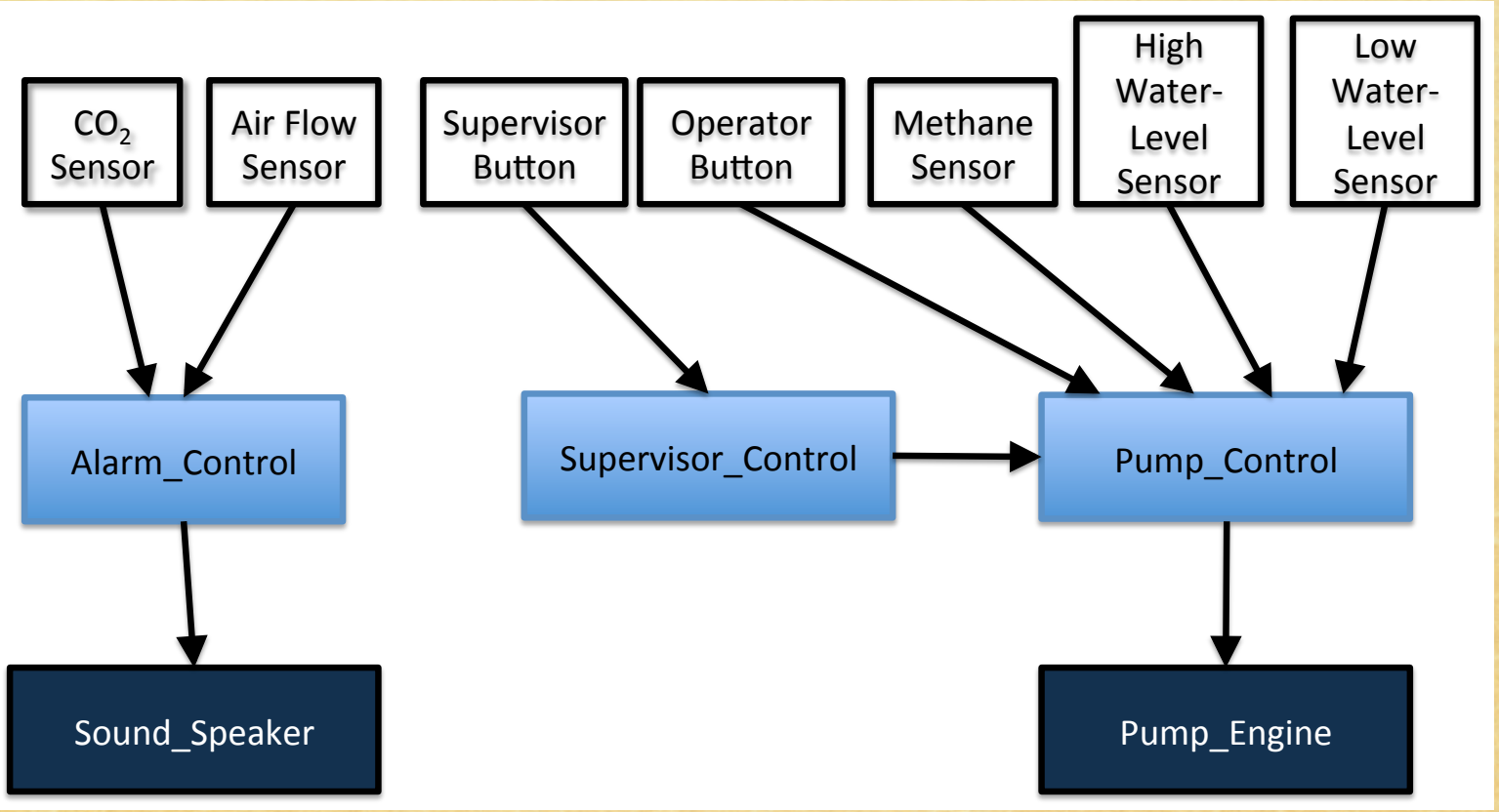


Figure 8: `Mine_Pump` Dependencies.

Comparison

Table 4: Resource comparison for Kripke structure of the `Mine_Pump` case study using `gufsm` and `NuSMV`as in Table 2.

Component	CPU Time	Space
Combined graph	22,356.51 s	2,611,097 Kb
Alarm subgraph	0.003 s	10 Kb
Supervisor subgraph	3.025 s	25,703 Kb

Summary

- Formal verification by model checking and the construction of FMEA tables had been reported to
  - take CPU times of the order of days or weeks for some well-discussed case studies (Grunske et al., 2011).
- We have shown here that, for logic-labeled FSMs,
  - we can efficiently split the corresponding dependency graph and
  - obtain components of the graph that can be analyzed independently.
- Consequently

  # of states of the system = $\prod$ *# of states of each subsystem*
  - replaced by

  # of states of the system = $\sum$ *# of states of each subsystem*

Conclusions

- With decomposition,
  - even only identifying two or three such components,
    - results in improvements in performance of **several orders of magnitude** for a single model- checking exercise
- Applicable particularly to loosely coupled systems
- Kripke structures in description languages of common tools such as NuSMV can be generated and verified much more efficiently.

Thanks

Questions