

Model Checking of Transition-Labeled Finite-State Machines

Vladimir Estivill-Castro¹ and David A. Rosenblueth²

¹ School of Information and Communication Technology
Griffith University

<http://vladestivillcastro.net>

² Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
<http://leibniz.iimas.unam.mx/~drosenbl/>

Abstract. We show that recent Model-driven Engineering that uses sequential finite state models in combination with a common sense logic is subject to efficient model checking. To achieve this, we first provide a formal semantics of the models. Using this semantics and methods for modeling sequential programs we obtain small Kripke structures. When considering the logics, we need to extend this to handle external variables and the possibilities of those variables been affected at any time during the execution of the sequential finite state machine. Thus, we extend the construction of the Kripke structure to this case. As a proof of concept, we use a classical example of modeling a microwave behavior and producing the corresponding software directly from models. The construction of the Kripke structure has been implemented using `flex`, `bison` and `C++`, and properties are verified using `NuSMV`.

Keywords: Model-driven engineering, embedded software, Model-checking, Kripke structures, sequential finite-state machines, common sense logics.

1 Introduction

Model-driven engineering (MDE) is a powerful paradigm for software deployment with particular success in embedded systems. The domain models created are intended to be automatically converted into working systems, minimizing the potential faults introduced along the more traditional approaches that translate requirements into implementation. MDE offers traceability of the requirements to implementation, enabling validation and verification. Because software modeling is the primary focus, and there is an emphasis in automation, it has been suggested [15] that MDE can address the inabilities of third-generation languages to alleviate the complexity of releasing quality software for diverse platforms and to express domain concepts effectively.

Recently, a flexible tool to model the behavior of autonomous robotics systems has been finite state machines where transitions are labeled with statements that must be proved by an inference engine [3]. The modeling capability of these Transition-Labelled Finite State Machines (FSMs) has been shown [2] to be remarkably expressive with respect to other paradigms for modeling behavior, like

Petri Nets, Behavior Trees and even standard finite-state machines like those of executable UML [11] or StateWorks [17]. If models of behavior are to be automatically converted into deployed systems, it is of crucial importance to verify the correctness of such models. Model checking promises to precisely enable this. Our aim here is to provide efficient and practical model-checking techniques to verify these new sequential finite state machines. We define one path to enable the model-checking of the behavior of robots expressed as finite state machines. We will describe the Finite-State Machines with transitions labeled by Boolean expressions and we will provide a semantics for it by describing the behavior in terms of sequential programs [7, Section 2.2]. Sequential programs are converted into first-order representations [7, Section 2.1.1] and these into Kripke structures. The technology for verifying properties in Kripke structures is well established. This step is perhaps direct since logics to express properties of the model are well understood as well as the algorithms to perform the validation and there are solid implementations. We will be concerned with only the sequential part of the behavior.

1.1 Sequential finite-state machines

An important sub-class of our models are sequential finite-state machines which we can represent visually to facilitate their specification, but which have a precise semantics. In fact, our sequential finite-state machines are a language for sequential programs and therefore, a natural approach is to describe their semantics as sequential programs [10].

A sequential finite-state machine consist of the following elements:

1. A set S of *states*, one of which is designated as the initial state $s_0 \in S$.
2. Each state s_i has associated with it a finite (and possibly empty) list $L_i = \langle t_{i1}, t_{i2}, \dots, t_{i, it} \rangle$ of *transitions*. A transition t_{ij} is a pair (e_{ij}, s_j) , where e_{ij} is a Boolean expression (a predicate that evaluates to **true** or **false**) and s_j is a state (i.e. $s_j \in S$) named the *target* of the transition. For all the transitions in L_i , the state $s_i \in S$ is called the *source*.
3. Each state has three *activities*. These activities are labeled **On-Entry**, **On-Exit**, and **Internal**, respectively. An activity is either an *atomic statement* P (and for the time being the only atomic statement is the assignment $x := e$) or a compound statement $P = \langle P_1; P_2; \dots P_t \rangle$ where each P_k is an atomic statement.

Note that sequential finite-state machines have a very similar structure to UML's state machines [11] and OMT's state diagrams [14, Chapter 5].

Because of its structure, a sequential finite-state machine can be encoded by two tables. The first table is the *activities table* and has one row for each state. The columns of the table are the state identifier, and columns for the **On-Entry**, **On-Exit**, and **Internal** activities. The order of the rows does not matter except that the initial state is always listed as the last. The second table of the sequential finite-state machine is the *transitions table*. Here, the rows are triplets

of the form (s_i, e_{ij}, s_j) where s_i is the source state, e_{ij} is the Boolean expression and s_j is the target state. In this table, all rows for the same source state must appear, for historical reasons, in the reverse order than that prescribed by the list L_i . Sequential finite-state machines can also be illustrated by state diagrams.

1.2 The corresponding sequential program

The intent of a sequential finite-state machine model is to represent a single thread of execution, by which, on arrival to a state, the **On-Entry** statement (atomic or compound) is executed. Then, each Boolean expression in the list of transitions out of this state is evaluated. As soon as one of them evaluates to **true**, we say the transition *fires*, and the **On-Exit** statement is executed, followed by repeating this execution on the target state. However, if none of the transitions out of the state fires, then the **Internal** activity is performed followed by the re-evaluation of the outgoing transitions in the corresponding list order.

Note again the importance of the transitions out of a state being structured in a list. This means that the sequential finite-state machine is not the same as the finite-state automaton usually defined in the theory of computer languages. In particular, there is no need for the Boolean expression to be exclusive. That is, $e_{ij} \wedge e_{ij'}$ may be **true** for two different target states s_j and $s_{j'}$ out of the same source state s_i . Also, there is no need for the Boolean expressions to be exhaustive. That is $\bigvee_{j=1}^{j=it} e_{ij}$ may be false (and not necessarily true).

The procedure in Fig. 1 provides an operational semantics of a sequential finite-state machine viewed as sequential program.

1.3 From sequential programs to Kripke structures

Thus, a sequential finite-state machine is just a special sequential program and we can apply a transformation [7] to obtain its corresponding Kripke structure, which can be verified. The crucial element is to make an abstraction identifying all possible breakpoints in a debugger for the program. These break points are actually the values for a special variable *pc* called the *program counter*.

However, what is the challenge? Under the transformation \mathcal{C} [7, Section 2.2], the number of states of the Kripke structure grows quickly. For example, the direct transformation of a sequential finite-state machine with three states, two Boolean variables and four transitions results in $2^2 \times 3^2 \times 2 \times 2 \times 20 = 2,880$. Therefore, an automatic approach to generate the Kripke structure is needed and perhaps more interestingly, an alternative approach that can perhaps obtain a more succinct and efficient approach. Observe, however, that we only need to consider as break points (labels) the following points in the execution:

1. after the execution of the **OnEntry** activities in the state,
2. after the evaluation of each Boolean expression labeling the transitions,
3. after the execution of the internal activities of the state, and
4. after the execution of the **OnExit** activities of the state (which corresponds to a break point before the **OnEntry** activities of the next state).

```

current_state ← s0; {Initial state is set up}
fired ← true ; {Default arrival to a state is because a transition fired}
{Infinite loop}
while ( true ) do
  if ( fired ) then
    {On arrival to a state execute On-Entry activity}
    execute ( current_state.on_Entry ) ;
  end if

  {If the state has no transitions out halt}
  if (  $\emptyset$  == current_state.transition_List ) then
    halt;
  end if
  {Evaluate transitions in order until one fires or end of list}
  out_Transition ← current_state.transition_List.first;
  fired ← false;
  while ( out_Transition ≤ current_state.transition_List.end AND NOT fired ) do
    if ( fired ← evaluate (current_state.out_Transition) ) then
      next_state ← current_state.out_Transition.target;
    end if
    out_Transition ← current_state.transition_List.next;
  end while
  {If a transition fired, move to next state, otherwise execute Internal activities}
  if ( fired ) then
    execute ( current_state.on_Exit ) ;
    current_state ← next_state;
  else
    execute ( current_state.Internal ) ;
    fired ← false;
  end if
end while

```

Fig. 1: The interpretation of a sequential finite-state machine.

Now, by exploring all the execution paths, we obtain the corresponding Kripke structure. If the machine has $\|V\|$ variables, and the largest domain of these is d , then the interpreter for the sequential finite state machine can be adjusted to a traversal in the graph of the Kripke structure. The number of nodes of the graph is the number of states of the Kripke structure and will be $4 \times \|V\|^d$.

2 Building the Kripke structure for NuSMV

The conversion of the model of a sequential FSM (or equivalently the transitions table and the activities table) into a Kripke structure description conforming to the NuSMV [5] syntax is as follows.

Generation rule 1 *Input files for NuSMV start with the line `MODULE main`, thus this is printed automatically. Then, a variables section starts with the keyword `VAR`, and here every variable used in the FSM is declared together with its range of values. Moreover, a variable `pc` standing for program counter is declared. The `pc` variable has a discrete range made of the following. For each state (listed in the activities table) with name `NAME`, there will be at least two values in the domain of the variable `pc`; namely `BEFORENAME`, and `AFTERONENTRYNAME`.*

The `BEFORENAME` value of `pc` corresponds to when the sequential program is about to commence execution of activities corresponding to the state `NAME`,

(and also to the departure of execution from the previous state, so there is no need for a value for `pc` equal to `AFTERONEXITNAME`).

The `AFTERONENTRYNAME`. value of `pc` corresponds to when the sequential program has completed execution of the **OnEntry** activity of the state `NAME`.

Also, for each transition out of state `NAME` labeled with a Boolean expression B_i the range of `pc` will include the values `AFTEREVALUATEBINAMETRUE` and `AFTEREVALUATEBINAMEFALSE`.

Generation rule 2 *The initial states of the Kripke structure are specified in a section labeled `INIT` by a predicate holding exactly at such states.*

Generation rule 3 *If the sequential FSM has more than one initial state and it can start at any of these, then the `INIT` section of the Kripke structure will have a disjunction*

$$\text{pc} = \text{BEFORES1} \mid \text{pc} = \text{BEFORES2},$$

indicating that all Kripke states where the `pc` is before an initial state of the sequential FSM are initial states of the Kripke structure.

The Kripke structure corresponding to the sequential FSM is described by specifying its transition in the section `TRANS` of the NuSMV input. In particular, a **case** statement is used. Each entry in the case statement corresponds to a Boolean predicate that describes a state of the Kripke structure, such as:

$$\text{x} = 0 \ \& \ \text{y} = 1 \ \& \ \text{pc} = \text{BEFORESTART}$$

The transitions in the Kripke structure are indicated by using the NuSMV function `next` after a colon `:` specifying the case statement, and using the symbol `|` for ‘or’ to indicate alternative transitions. For example,

$$\begin{aligned} & \text{x} = 1 \ \& \ \text{y} = 0 \ \& \ \text{pc} = \text{BEFORESTART} \\ & : \ \text{next}(\text{x}) = 1 \ \& \ \text{next}(\text{y}) = 0 \ \& \ \text{next}(\text{pc}) = \text{BEFORESTART} \\ & | \ \text{next}(\text{x}) = 0 \ \& \ \text{next}(\text{y}) = 0 \ \& \ \text{next}(\text{pc}) = \text{AFTERONENTRYSTART}; \end{aligned}$$

describes two transitions in the Kripke structure: a self-loop in the state with $\text{x} = 1 \ \& \ \text{y} = 0 \ \& \ \text{pc} = \text{BEFORESTART}$ and a transition to the Kripke state $\text{x} = 0 \ \& \ \text{y} = 0 \ \& \ \text{pc} = \text{AFTERONENTRYSTART}$. The approach in [7, Section 2.1.1] always places a self-loop (that leaves all variables intact) to model the possible execution in a multi-tasking environment where the execution may be delayed indefinitely.

Generation rule 4 *For every Kripke state `NAME` where the `pc` has value `BEFORENAME`, there will be two departing transitions, one is the self-loop. The second transition will affect the variables by the execution of the **OnEntry** action and move the `pc` variable to `AFTERONENTRYNAME`.*

Generation rule 5 *A Kripke state with `pc=AFTERONENTRYNAME` will produce a self-loop and also another transition resulting of evaluating the first Boolean expression corresponding to the first transition in the sequential finite state machine. Because a Boolean expression is evaluated, none of the variables except `pc` changes value. If the Boolean expression evaluates to **true**,*

then the variable `pc` changes to `pc= AFTEREVALUATEB1STARTTRUE`; otherwise, when the Boolean expression evaluates to `false`, then `pc= AFTEREVALUATEB1STARTFALSE`.

Generation rule 6 *A Kripke state with `pc=AFTEREVALUATEBiNAMETRUE` will produce a self-loop and also another transition resulting of executing the **OnExit** statement of the state `NAME` of the sequential FSM. The target state in the Kripke structure of this transition will have the variables modified by the execution of the statement in the **OnExit** part and the variable `pc` set to a value `BEFORETARGET` where `TARGET` is the state of the sequential FSM that is the target of the transition that fired.*

Generation rule 7 *A Kripke state with `pc=AFTEREVALUATEBiNAMEFALSE` will produce a self-loop and if B_i is the Boolean expression, but for any other FSM-transition except the last transition, the second Kripke transition is the result of evaluating the next Boolean expression labeling the next FSM-transition.*

3 Applying Model-Checking to FSM+DPL

A classical example on modeling software behavior has been a microwave oven (e.g., [16]). This example also appears in the literature of requirements engineering (e.g., [8]). Recently, the modeling tool FSM+DPL that consists of finite-state machines whose transitions are labeled with queries to the inference engine of DPL has been applied to the microwave oven example, producing shorter and clearer models than Petri Nets, Behavior Trees or other approaches also using finite-state machines [2]. It was also shown that these models could be directly converted into executable implementations (using different target languages, in one case Java and in the other C++ and on independent platforms, in one case a Lego-Mindstorm and in the other a Nao robot). They have been effective in modeling behavior of robots for RoboCup 2011 [9].

We illustrate here that the FSM+DPL approach is also receptive to be formally verified using the machinery of model checking using Kripke structures. We use the generic methodology developed in the previous sections for sequential FSM, but we will require an extension to handle external variables. To illustrate the general approach, we reproduce some parts of a microwave model with FSM+DPL [3]. Here, a part of the model is the sequential finite-state machines. One of these FSM is illustrated in Fig. 2a. Associated with each of these machines is the corresponding code for a logic (formally a theory). The logic used is DPL [13], which is an implementation of a variant of *Plausible Logic* [1, 4, 12]. The theory for the engine, tube, fan and plate (see Fig. 2b) can be thought of as the declarative description of an expert on whether we should `cook` or `not cook`. These are labeled as outputs. The expert would like to have information about whether there is `timeLeft` and/or whether `doorOpen` is true or false (that is, whether the door is open or not). This is not absolutely necessary; however, these desirable inputs are labeled as such. The actual inference rules are described by rules. The implementation of DPL compiles (translates) the theory

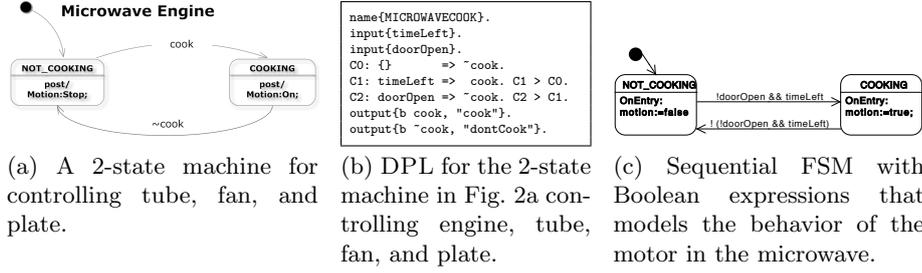


Fig. 2: Simple 2-state machines control most of the microwave.

to equivalent C Boolean expressions, or more simply to Boolean expressions. For example, using the the DPL tool with the `+c` option translates the theory about cooking in Fig 2b to `!doorOpen && timeLeft`. The corresponding finite-state machines are as in Fig. 2c.

Generation rule 8 *A FSM+DPL models is first translated to a sequential FSM with Boolean expressions in the transitions.*

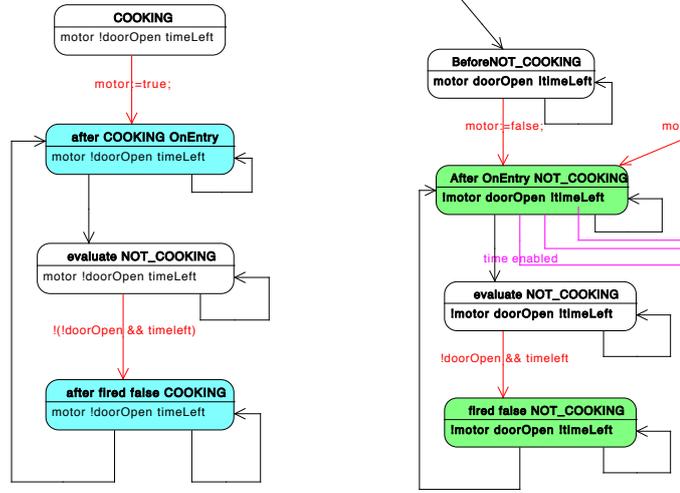
We now proceed to outline the transformation of this resulting sequential FSM into the corresponding Kripke structure. Here we have the Boolean variables `timeLeft`, `doorOpen`, and `motor`.

There is an important difference in that now `timeLeft` and `doorOpen` are *external variables*. That is, the sequential program equivalent to this sequential FSM cannot modify the values of these variables. Instead, it is an external agent who does. We model this using the tools of parallel computation illustrated also in [7, Section 2.1]. We will start with a Kripke structure where:

1. such a change to external variables cannot be undetected by the corresponding sequential program, and
2. computational statements in the **OnEntry**, **OnExit** or **Internal** activities and evaluation of Boolean expressions labeling transitions are atomic.

Now, recall (Rule 1) that for each state `NAME` in the sequential FSM we have a state in the Kripke structure named `BEFORENAME` for a valuation with `pc=BEFORENAME`. For each state of our sequential FSM, first we execute the assignment of the **OnEntry** component and then we enter an infinite loop where we evaluate a Boolean expression. When the variable `pc` is just after the execution of the **OnEntry** activity we have the value `AFTERONENTRYNAME`.

We will model the loop of the semantics of the sequential FSM state execution with a substructure in the Kripke structure we call a *ringlet*. This essentially corresponds to the alternation of two actions in the sequential program and four states of the Kripke structure (refer to Fig. 3) using the standard transformation [7]. Three of the states are named as before, but here we introduce one to handle the external agent affecting the variables before an atomic evaluation of the Boolean expression labeling a transition.



(a) One ringlet, representing the sequential program going through the loop of evaluating one state in the sequential state machine.

(b) Another ringlet, for state **NOT_COOKING** and valuation *motor*, *doorOpen*, *! timeLeft*

Fig. 3: Sections of the Kripke structure for the example of Fig. 2

BEFORENAME: The Kripke state corresponds to the label of the sequential program just before the execution of the programming statement in the **OnEntry** component.

AFTERONENTRYNAME: This Kripke state reflects the effect of the programming statement in the variables (in the case from Fig. 2a, there can only be one variable, the *motor* variable). In our diagrams, these Kripke states have a background color to reflect the arrival there by the execution of a statement in the sequential program.

BEFOREEVALUATIONBINAME: This state in the Kripke structure represents the same valuation for the variables. In a ringlet, it represents that after the previous programmed action of the sequential FSM the external agent did not affect the value of any of the external variables.

AFTEREVALUATIONBITRUE: As before, this Kripke state represents that the evaluation of the Boolean expression in the sequential FSM (and therefore in the sequential program) evaluated to true. This also does not change the valuation of the variables because it just evaluates a Boolean expression. It will also have a background color to indicate that the Kripke structure arrives here because of the action of the sequential FSM. From here the machine moves to perform the **OnExit** activities.

AFTEREVALUATIONBIFALSE: As before, this Kripke state represents that the evaluation of the Boolean expression in the sequential FSM (and therefore in the sequential program) evaluated to false. This also does not change the valuation of the variables because it just evaluates a Boolean expression.

It will also have a background color to indicate that the Kripke structure arrives here because of the action of the sequential FSM. If this is the last transition, it will go to perform the **Internal** activities; otherwise, it will go to the next AFTEREVALUATIONBITRUE.

Figure 3a illustrates the ringlet for the sequential FSM in the state of COOKING, in particular the moment when we commence with an assignment of the variables `motor` and `timeLeft` set to `true` but the variable `doorOpen` set to `false`. We have colored red the transitions resulting from the sequential program actions. We have also labeled the transition with the assignment or the Boolean expression in red; these labels are only comments. In Fig. 3a, this transition is the execution of `motor:=true`. Therefore, the second Kripke state has a new assignment for the values. Transitions appearing in black in the Kripke structure identify labels of the sequential program and are part of the construction of the Kripke structure from labeled sequential programs as before. A similar portion of the Kripke structure, for the sequential FSM state of NOT_COOKING and with an assignment of the variables `motor`, `doorOpen`, to `true`, but `timeLeft` to `false`, appears in Fig. 3b. Here, however, the programming statement that will be executed is `motor:=false`, and the transition guarding the state NOT_COOKING in the sequential FSM is `!doorOpen && timeLeft`. Thus, the final Kripke structure has at most $2 \times 4 \times 8 = 64$ states.

We now explain other transitions of the Kripke structure that model the external variables. Figure 4 illustrates part of the Kripke structure. It shows four of the eight initial states of the Kripke structure. Each initial state of the sequential FSM should be the first Kripke state of a four-state ringlet. However, some Kripke structure states for the NOT_COOKING state of the sequential FSM do not have a ringlet. This is because the assignment in the **OnEntry** part modifies the value of `motor` to `false`, making the ringlet impossible.

Now, after the execution of the assignment statement and also after the evaluation of the Boolean expression by the sequential program, the external variables `doorOpen` and `timeLeft` may have been modified.

We will model the modification of external variables with magenta transitions outside a ringlet before the evaluation of Boolean expression in sequential FSM. One such transition links an AFTERONENTRY Kripke state to a BEFOREEVALUATION Kripke state. In this way, the sequential FSM must traverse a ringlet at least once and notice the change of the external variables. Therefore, there will be three transitions out of the second state of every ringlet in the Kripke structure. This corresponds to:

1. the value of `doorOpen` being flipped (from `true` to `false` or vice versa),
2. the value of `timeLeft` being flipped, and
3. the values of both `doorOpen` and `timeLeft` being flipped.

In Fig. 4 we only show these transitions out of the leftmost ringlet.

The above description shows that we can construct a Kripke structure with no more than 64 states that completely captures the model of sequential FSM

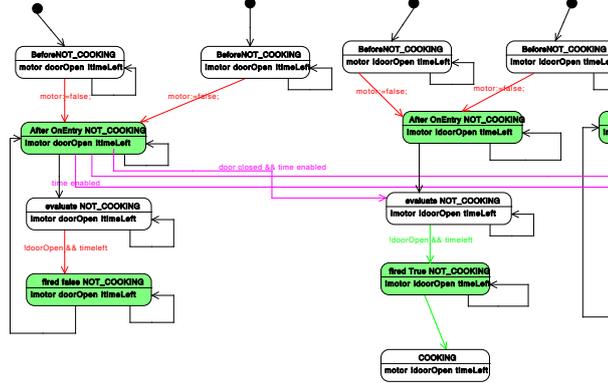


Fig. 4: Partial view of the Kripke structure for the Sequential FSM with external variables of Fig. 2c.

with transitions labeled by DPL and that involve external variables. To automatically construct such Kripke structures we have implemented a C++ program that uses `flex` and `bison` to scan and parse the activities and transitions table of a sequential finite-state machine. As output, it produces the corresponding Kripke structure in a format suitable for NuSMV.

3.1 The model-checking case study

To complete the demonstration that the FSM+DPL model-driven approach can be subject to model-checking by our proposal here, we now discuss some properties verified using our implementation of the constructor of the derived Kripke structure in combination with NuSMV.

In particular, an important property is the safety requirement that “*necessarily, the oven stops three transitions in the Kripke structure after the door opens*”. In CTL (Computation-Tree Logic [6]) this is the following formula:

$$\text{AG}((\text{doorOpen}=1 \ \& \ \text{motor}=1 \ \rightarrow \ \text{AX} \ \text{AX} \ \text{AX}(\text{motor}=0)) \quad (1)$$

Another illustrative property for which NuSMV return favorable verification is the following. “*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*”. The CTL formula follows.

$$\text{!E}[\text{!(doorOpen=0)U}(\text{motor}=1 \ \& \ \text{!(pc=BeforeNOT_COOKING)})] \quad (2)$$

State ID	On-Entry	On-Exit	Internal
COOKING	motor:= false;	∅	∅
NOT_COOKING	motor:= true;	∅	∅

Table 1: Faulty state table. The statements on the **OnEntry** sections are in the wrong state with respect to the correct model in Fig. 2c.

Observe that these properties can be used for effective verification of the model. For example, Table 1 is an erroneous table for the model of Fig. 2c since (by accident or as a result of a programming mistake) the assignments in the states of the sequential FSM are in the opposite state. This makes both Property (1) and Property (2) false in the derived Kripke structure. Also, in the second one, the reader may find it surprising that we have a condition on the `pc`. However, if this condition is removed, NuSMV will determine that the property is false, and will provide as a trace an initial state of the Kripke structure. The reason is that the Kripke structure construction assumes that the FSM can commence execution in any combination of the variables, and of course, one such combination is to have the door closed and the motor on. However, the property proves that if the sequential FSM is allowed to execute, then it will turn the motor off and from then on, the motor will only be on when the door is closed.

Another important property is the safety requirement that “*necessarily, the oven stops three transitions in the Kripke structure after the time elapses*”. In CTL this is the following formula.

$$\text{AG}((\text{timeLeft}=0 \ \& \ \text{motor}=1) \rightarrow \text{AX AX AX}(\text{motor}=0)) \quad (3)$$

This property is equivalent to “*the microwave will necessarily stop whether the door opens or the time expires, immediately after the `pc` advances no more than one ringlet*”.

Certain properties also show the robustness of the model. A property like “*cooking may go on for ever*” (coded as $\text{AG}(\text{motor}=1 \rightarrow \text{EX motor}=1)$) is false; naturally, because opening the door or the time expiring halts cooking. However,

$$\begin{aligned} &\text{AG}(\text{doorOpen}=0 \ \& \ \text{timeLeft}=1 \ \& \ \text{motor}=1 \\ &\quad \& \ !(\text{pc}=\text{BeforeNOT_COOKING})) \\ &\quad \rightarrow \text{EX}(\text{doorOpen}=0 \ \& \ \text{timeLeft}=1 \ \& \ \text{motor}=1) \end{aligned}$$

indicates that, from a state which is not an initial state, as long as the door is closed, and there is time left, cooking can go on for ever.

4 Final remarks

Our Kripke structure conversion is efficient in that, if the sequential finite state machine has n states and an average of m transitions per state, then our Kripke structure has a number of Kripke states bounded by $(4n + m)f(k)$. That is, the Kripke structure complexity is linear in the number of states and the number of transitions of the sequential FSM. The potential complexity challenge is on the number of the variables used in the sequential FSM (both internal and external). This is because the function $f(k)$ can be exponential in k where k is the number of variables involved. In a sense, this has the favor of fixed-parameterized complexity, with the number of variables as the parameter. The number of transitions in the Kripke structure is also linear in n and m , but potentially exponential in the number of external variables. Hence, the Kripke structure has a number of transitions linear in its number of states, so that the Kripke structure as a graph

is rather sparse. In summary, the models produced by this approach could be considered efficient.

Acknowledgments. We thank many colleagues from the Mi-PAL team that have enabled the design and implementation of many facilities that constitute the infrastructure to enable Transition-Labelled Finite State Machines into a practical modeling-driven engineering method for developing behaviors on autonomous mobile robots. We also thank Miguel Carrillo for fruitful discussions on the microwave oven example, as well as the facilities provided by IIMAS, UNAM. DR was supported by grant PAPIIT IN120509.

References

1. D. Billington. The proof algorithms of plausible logic form a hierarchy. In S. Zhang and R. Jarvis, editors, *Proc 18th Australian Joint Conference on Artificial Intelligence*, volume 3809, pages 796–799, Sydney, Australia, December 2005. Springer Verlag Lecture Notes in Artificial Intelligence.
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)*, pages 68–77, Athens, Greece, 22-24 July 2010. SciTePress — Science and Technology Publications (Portugal).
3. D. Billington, V. Estivill-Castro, R. Hexel, and R. Rock. Modelling behaviour requirements for automatic interpretation, simulation and deployment. In *SIMPAR 2nd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Computer Science*, pages 204–216. Springer, 2010.
4. D. Billington and A. Rock. Propositional plausible logic: Introduction and implementation. *Studia Logica*, 67:243–269, 2001. ISSN 1572-8730.
5. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Int. J. on Software Tools for Technology Transfer*, 2:2000, 2000.
6. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, pages 52–71, IBM Watson Research Center, 1981. Lecture Notes in Computer Science No. 131.
7. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.
8. R. G. Dromey and D. Powell. Early requirements defect detection. *TickIT Journal*, 4Q05:3–13, 2005.
9. V. Estivill-Castro and R. Hexel. Module interactions for model-driven engineering of complex behavior of autonomous robots. In P. Dini, editor, *ICSEA 6th Int. Conf. on Software Engineering Advances*, Barcelona, Oct. 2011. IEEE. to appear.
10. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.
11. S. J. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture*. Addison-Wesley Publishing Co., Reading, MA, 2002.
12. A. Rock and D. Billington. An implementation of propositional plausible logic. In *23rd Australasian Computer Science Conference (ACSC 2000)*, pages 204–210. IEEE Computer Society, 31 January - 3 February 2000.
13. Andrew Rock. The DPL (decisive Plausible Logic) tool. Technical report, (continually) in preparation, available at www.cit.gu.edu.au/~arock/.

14. J. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modelling and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
15. D.C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), 2006.
16. S. Shlaer and S. J. Mellor. *Object lifecycles : modeling the world in states*. Yourdon Press, Englewood Cliffs, N.J., 1992.
17. F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, NY, 2006.