Efficient Construction of Kripke Structures and Model Checking of Logic-Labeled Sequential Finite State Machines

Author
Estivill-Castro, Vladimir, A. Rosenblueth, David

Published
2013

Journal Title
Information

Downloaded from
http://hdl.handle.net/10072/55403

Link to published version
http://www.information-iii.org/abs_e2.html#No2%28B%29-2013
Efficient Construction of Kripke Structures and Model Checking of Logic-Labeled Sequential Finite State Machines

Vladimir Estivill-Castro* and David A. Rosenblueth**

*School of Information and Communication Technology
Griffith University
http://vladestivillcastro.net

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

Abstract

We show that Model-driven Engineering using sequential finite-state models, in combination with a common sense logic is subject to efficient model checking. To achieve this, we first provide a semantics of the models. Using this semantics and methods for modeling sequential programs we obtain small Kripke structures. 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 the NuSMV model checker.

Keywords: Model-driven Engineering, Model checking, Finite-state machine, Kripke structure.

1 Introduction

Model-driven engineering is a powerful paradigm for software deployment with particular success in embedded systems. 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, executable UML and StateWorks. 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 sequential finite-state machines.

1.1 Sequential finite-state machines

Our models are sequential finite-state machines that can be visually represented to facilitate their specification, but which have a precise semantics [7].
A sequential finite-state machine consists 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 \( L_i = \{t_{i1}, t_{i2}, \ldots, t_{i|L_i|}\} \) of transitions. A transition \( t_{ij} \) is a pair \((e_j, s_j)\), where \( e_j \) is a Boolean expression and \( s_j \) is a state (i.e. \( s_j \in S \)) named the target of the transition. 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 \( P = \langle P_1; P_2; \ldots; P_l \rangle \) where each \( P_k \) is an atomic statement.

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 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 \((s_i, e_j, s_j)\) where \( s_i \) is the source state, \( e_j \) is the Boolean expression and \( s_j \) is the target state.

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 on 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.

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 [6] to obtain its corresponding Kripke structure, which can be verified.

However, what is the challenge? Under the transformation \( \mathcal{C} [6, \text{Section 2.2}] \), 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 \( 2^2 \times 3^2 \times 2 \times 2 \times 4 = 2,880 \). Therefore, an alternative approach is needed, to obtain a more succinct Kripke structure. Observe 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).

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 a variable-declaration section, where every variable used in the FSM (as well as pc) is declared together with its range of values. 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.

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

$$pc = \text{BEFORE}S1 \mid pc = \text{BEFORE}S2,$$

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

Generation rule 4 For every Kripke state NAME where the pc 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=AFTEREVALUATEB1NAMETRUE 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 appearing both in software-behavior modeling and requirements engineering has been a microwave oven. Recently, the modeling tool FSM+DPL has been applied to the microwave oven example, producing shorter and clearer models that Petri Nets, Behavior Trees or other approaches also using finite-state machines [2].

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. One of the FSMs is illustrated in Fig. 1a. Associated with each of these machines is the corresponding code for a logic (formally a theory). The logic used is DPL, which is an implementation of a variant of Plausible Logic [1, 4]. The theory for the engine, tube, fan and plate (see Fig. 1b) 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. The corresponding finite-state machines are as in Fig. 1c.

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. We model this using the tools of parallel computation illustrated also in [6, Section 2.1].

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. 2) using the standard transformation [6]. 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.

This construction [7] builds a Kripke structure with no more than 64 states that completely captures the model of sequential FSM 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.

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:

\[ AG( \text{doorOpen}=1 \land \text{motor}=1 \rightarrow AX \ AX \ AX(\text{motor}=0) ) \]

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, $\text{AG}(\text{doorOpen}=0 \& \text{timeLeft}=1 \& \text{motor}=1 \& \neg \text{(pc=Before\_NOT\_COOKING)})$ 

$\rightarrow \text{EX}(\text{doorOpen}=0 \& \text{timeLeft}=1 \& \text{motor}=1)$

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. We refer the reader to [7] for more properties.

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 $(4n + m)f(k)$.

Acknowledgments

We should thank the Mi-PAL team as well as the facilities provided by IIMAS.

References


