International Journal of Networked and Distributed Computing, Vol. 4, No. 1 (January 2016), 65-74

Using SPIN to Check Simulink Stateflow Models

Chikatoshi Yamada [1], D. Michael Miller [2]

[1] *Department of Information Communication Systems Engineering, National Institute of Technology, Okinawa College, 905 Henoko, Nago-shi, Okinawa 905-2192, Japan*

E-mail: cyamada@okinawa-ct.ac.jp

[2] *Department of Computer Science, University of Victoria, PO Box 3055, Victoria, British Columbia V8W 3P6, Canada*

E-mail: mmiller@uvic.ca

Abstract

Verification is critical to the design of large and complex systems. SPIN is a well-known and extensively used verification tool. In this paper, we consider two tool chains, one existing, WSAT, and one introduced here, that support using SPIN to model check systems specified as Simulink Stateflow models. We present algorithms for doing the necessary translations and present empirical results that show the chain using tools introduced in this paper performs better than the one using the existing WSAT tool. We also show that these tools allow SPIN to be used for model checking nondeterministic Stateflow models in addition to deterministic ones.

Keywords: Model checking, Simulink Stateflow, SPIN, tool chains.

1. Introduction

Verification (model checking) plays an important role in the design of large scale and complex systems. The technique is applied to software requirement specifications and design specifications, and aims to increase reliability and productivity from the early stages of software development. System verification ascertains whether designed systems can be executed or specified. Various formal methods for verification have been studied [1,2,3].

The objective of the work described in this paper is to provide a tool chain that supports using the well-known tool SPIN [4] to model check systems specified as Simulink Stateflow models [5]. As shown in Fig. 1, we consider two such tool chains. In the first chain, XML produced by Simulink is translated via an intermediate file format to MSL using new tools described later in this paper. WSAT [6] is used to translate the MSL code to PROMELA for input to SPIN. In the second chain, another new tool, also described later in this paper, translates the intermediate file found from the Simulink produced XML directly to PROMELA. Results given later in the paper show the second approach performs better. We also give an example of a nondeterministic model that cannot be checked in Simulink that can be checked using SPIN which shows the diversity of application of our new tools.

The remainder of the paper is organized as follows. The requisite background is reviewed in Section 2 and related work is described in Section 3.

The tools introduced in this paper are described in Section 4. Experimental results are presented in Section 5 and the paper concludes with our observations and suggestions for future work in Section 6.

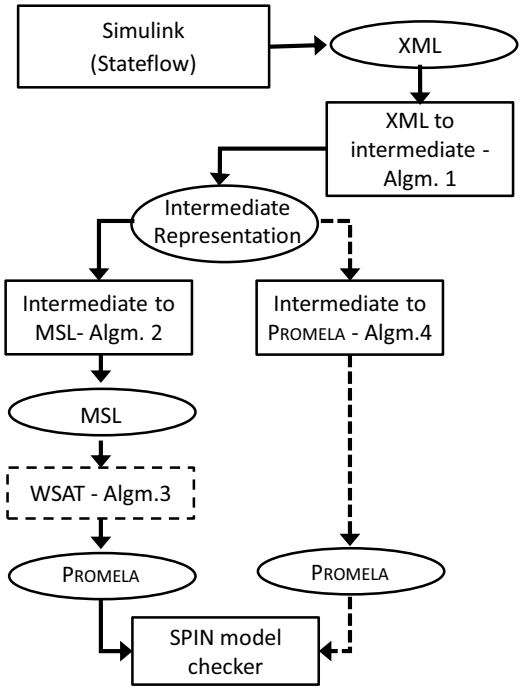

Fig. 1. Tool chains considered in this paper.

2. Background

Model checking is a process that explores a finite state space to determine whether or not a given property holds. The common properties considered are:

- **Reachability**: indicates that some particular situation, or situations, can be reached.
- **Safety**: indicates that the system can not transition to an invalid state.
- **Fairness**: indicates that any process than can execute an action will eventually be able to execute that action.
- **Liveness**: indicates that a process can always eventually reach a desired state.

The following tools and languages are used in this work:

Simulink [5] is a graphical programming tool developed by MathWorks for modeling, simulating and analyzing multidomain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It integrates with the MATLAB environment and can either drive MATLAB or be scripted from it allowing one to use MATLAB functionality in modeling and analysis.

Simulink includes a Design Verifier [7] that uses formal methods to identify design errors in models including dead logic, integer overflow, division by zero, and violations of design properties and assertions. It does not test for liveness, reachability or deadlock [8].

Extensible Markup Language (XML) [9] is a widely used markup language that defines a set of rules for encoding information in a format that is both human-readable and machine-readable, and that is easily exchanged. Simulink provides a utility to express a Stateflow model in XML.

Model Schema Language (MSL) [10] is a formalism of the core ideas in XML schema.

Web Service Analysis Tool (WSAT) [11,6,?] is a formal specification, verification, and analysis tool for web service compositions. The tool supports multiple specification approaches of composite web services. In addition, WSAT provides several analyzes techniques that can deduce complexities in the formal verification of asynchronously communicating web services. WSAT translates web service input in MSL into PROMELA thereby supporting the use of SPIN.

Process or Protocol Meta Language (PROMELA) is a verification modeling language that allows for the dynamic creation of concurrent processes to model various types of systems. PROMELA programs consist of processes, message channels, and variables. Processes are global objects that represent the behaviour of concurrent entities of a distributed system. Channels and global variables define the environment in which the processes run. PROMELA is used to describe systems to be modeled using SPIN.

Simple PROMELA Interpreter (SPIN) [12,4] is a general tool designed to verify the correctness of distributed software models in a rigorous fashion. PROMELA is used to describe the system to be modeled. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user. SPIN gains efficiency by generating optimized C code modules that perform the requested model checks.

3. Related Work

3.1. Simulink model checking

Model checking for Matlab/Simulink models has been researched in [13,8,?]. In [13], the authors describe a translation process that can convert a well-defined subset of MSS (Matlabs Simulink and Stateflow) into a standard form of hybrid automata. In his thesis[8], Leitner compares Simulink Design Verifier and the SPIN model checking tool. The comparison is based on the case study of an AUTOSAR compliant memory management module. In this context, it is also described how Simulink/Stateflow models can be manually translated into the input language of the model checker SPIN. In [14], the authors report their initial experience in using model-based safety analysis on an example system taken from the ARP Safety Assessment Guidelines document.

3.2. UML/Statechart to PROMELA

The basic ideas for the translation of UML/Statechart to PROMELA have been well-defined in [15,16]. In[15], the translator is an implemented system, integrated with a commercial design tool. In their implementation they use Visio as the front-end GUI, which connects to Visual Basic to execute the translator. Reference [16] presents a translation from a subset of UML Statechart Diagrams - covering essential aspects of both concurrent behaviour, like sequentialization, parallelism, non-determinism and priority, and state refinement - into PROMELA. The work reported in [16] provides basic ideas regarding translation from UML Statecharts to PROMELA.

3.3. C to PROMELA

Translation from other language to PROMELA, have been developed in [17,18,?]. In [17], Jiang addresses the problem of automatically verifying the correctness of concurrent algorithms, and describes a step in this direction: an automated translation from a subset of C to PROMELA. Reference [18] gives a method for using the tool SPIN to verify the network protocol stack TCP/IP for communications. The approach consists of building a model of the underlying operating system to be joined with the original C code in order to obtain the input for the model checker. Reference[19] uses static analysis to automatically construct an abstract matching function which depends on the program and the property to be verified.

3.4. XML to PROMELA

Simulink models can produce XML (eXtensive Markup Language). A translator from XML to PROMELA has been developed as the WSAT tool [11,?,?]. The presented algorithms translate (bounded) XML data and XPath expressions to PROMELA. The techniques constitute the basic of their Web Service Analysis Tool (WSAT) which verifies LTL properties of composite web services. MSL, a compact formal model of XML, has a straightforward mapping to PROMELA. Boolean XPath expressions are used in branch for loop conditions, and location path and arithmetic expressions are used on the left and right hand sides of assignment statements. In this paper, we use basic ideas of translation XML to PROMELA but WSAT tool is different target areas.

4. Our Tools

Simulink is a well defined tool for Model-Based Development (MBD). We focus on the Simulink Stateflow model. A well-known Simulink cruise control demonstration example is shown as Fig.2.

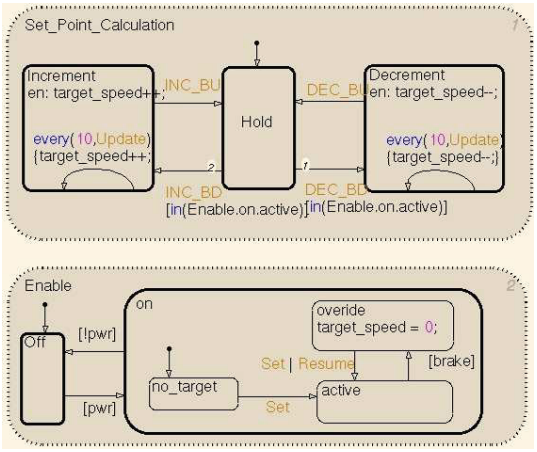

Fig. 2. Simulink stateflow model of a cruise control.

```
13:         end if
14:     end for
15:     for k ← 0, trN do
16:         if (tree is "transition") then
17:             tr_k ← "transitionID"
18:             tr_k.lb ← "transitionLabel"
19:         end if
20:         if ("src" is "state" or "junction") then
21:             tr_k.src ← ("state" or "junction")
22:         end if
23:         if ("dst" is "state" or "junction") then
24:             tr_k.dst ← ("state" or "junction")
25:         end if
26:     end for
27:     return (st, jc, tr)
28: end procedure
```

Fig. 3. XML to intermediate file

4.1. *Extracting states and transitions from XML descriptions*

Simulink has a function for producing XML from MDL which is a text description of a Stateflow model. XML can also be created from SLX, a binary format for Simulink models available from version R2012a onward.

The first step in our tools is to create an intermediate format file from the XML file shown as Fig.3. For example the file segment in Fig. 2 yields the list in Fig. 4. The intermediate file in Fig.4 identifies the states, junctions and transitions. Our extractor is implemented using Python, see Fig.3.

Fig.3 performs the translation from XML to a finite state machine intermediate specification for a deterministic or nondeterministic model. Line 3 shows components of the model such as the state st_i, junction jc_j and transition tr_k. Note that the structures have XML attributes from Python's lxml library. In line 7, 12 and 17, component "ID"s can be extracted as st_i("stateID"), jc_j("junctionID") and tr_k("transitionID"). In line 7 and 18, component "Label"s can be extracted as $st_i.lb$("stateLabel") and $tr_k.lb$("transitionLabel"), respectively. In line 21 and 24, $tr_k.src$ and $tr_k.dst$ indicate transition relations as $tr_k(tr_k.src, tr_k.dst)$.

```
%token state5
%token state4
%token state3
     :
trans19 ( state5  ->  state5 ) null
trans18 ( state4  ->  state4 ) null
trans11 ( null    ->  state3 ) "Set"
trans17 ( state5  ->  state3 ) "[brake]"
trans16 ( state4  ->  state3 ) "Set | Resume"
```

Fig. 4. Intermediate listing for code in Fig. 2.

4.2. XML to PROMELA using WSAT

WSAT (Web Service Analysis Tool) [11,?,?] is a tool for analyzing and verifying composite web service designs, with state of the art model checking techniques. WSAT can translate MSL to PROMELA code in order to use SPIN. However, because of specific protocol models for Web services, WSAT cannot directly translate an XML file of a Simulink model to PROMELA. We have thus developed a translator from our intermediate file format to MSL using Python LEX and YACC [20], see Fig.5.

In Fig.5, the set of *state* or *junctions* is extracted in line 5. Then *transitions* are extracted in line 10. Thus, the MSL contains all states, junctions and transitions.

Our implementation of the WSAT algorithm [11] is shown in Fig.6. Figs.3, 5 and 6 together provide a path from XML to PROMELA.

Fig.6 describes a procedure MSLToPROMELA. The first string, i.e., ret1, contains the type declaration for g. The second output is the attribute definitions for g, if g is an intermediate type. As shown in Fig.6, the function body of MSLToPROMELA translates the input MSL type declaration recursively according to the syntax rules.

```
1:  procedure INTERTOMSL(inter)
2:      string m1,m2
3:      switch case
4:      g → b:
5:      if (b is g("state"or"junction")) then
6:          m1="states"+"{"+g0, g1, ..., gk+"}"
7:          m2=null
8:      end if
9:      g → gn:
10:     if (gn is "transition") then
11:         m1=gn +"->"
12:         m2=gn+1, gn+2, ..., gn+k
13:     end if
14:     end switch
15:     return (m1, m2)
16: end procedure
```

Fig. 5. Intermediate file to MSL

```
1:  //ret1: type declaration for g, including intermediate types.
2:  //ret2: attribute definition for the input g if g is intermediate.
3:  procedure MSLTOPROMELA(MSL)
4:      string ret1, ret2
5:      switch case
6:      g → b:
7:        ret1=null
8:        ret2=b+" "+b+"value"
9:      g → t[g0]:
10:     if (g0 is an intermediate type) then
11:         // type: generate a unique name
12:         ret1=tr(g0)[1]+"typedef"+type+"{"+tr(g0)[2]+"}"
13:         ret2=type+" "+t
14:     else
15:         ret1=tr(g0)[1]+"typedef"+t+"{"+tr(g0)[2]+"}"
16:         ret2=null
17:     end if
18:     g → g1{m,n}:
19:       ret1=tr(g1)[1]
20:       ret2=tr(g1)[2]+" [ "+n+" ] "+"int"+g1.tag+"occ"
21:     g → g1, g2, ..., gk:
22:       ret1=tr(g1)[1]+tr(g2)[1]+ ... +tr(gk)[1]
23:       ret2=tr(g1)[2]+tr(g2)[2]+ ... +tr(gk)[2]
24:     g → g1 | g2 | ... | gk:
25:       ret1=tr(g1)[1]+tr(g2)[1]+ ... +tr(gk)[1]+"mtype{"+"m_"+g1.tag+ ... +"m_"+gk.tag+"}"
26:       ret2=tr(g1)[2]+tr(g2)[2]+ ... +tr(gk)[2]+"mtype choice"
27:     end switch
28:     return (ret1, ret2)
29: end procedure
```

Fig. 6. MSL to PROMELA - WSAT [11]

4.3. XML to PROMELA directly

The second tool chain we consider also uses Fig.3 to translate XML to our intermediate file format. We then use Fig.7 to translate the intermediate file to PROMELA directly. It employs Python LEX and YACC.

This approach is new and is a major contribution

of this paper as the experimental results will show. In Fig.7 we employ simpler comparisons, as shown in lines 7, 11 and 15, than those used in Fig.6.

```
1:  // f(st_i, jc_j, tr_k) : components from intermediate file
2:  // m1: type declaration for f.
3:  // m2: attribute definition for f.
4:  procedure INTERTOPROMELA(f(st_i, jc_j, tr_k))
5:      string m1, m2
6:      f → b: // input "state" and "junction"
7:      if (b is f("state") or f("junction")) then
8:          m1="mtype {"+f_0, f_1, ..., f_i+"}"
9:          m2=null
10:     end if
11:     if (f_n is f("machine")) then
12:         m1="proctype"+"machinename"+"()"
13:         m2="transitions"
14:     end if
15:     if (f_n is f("transition")) then
16:         m1="atomic {" f_n +"->"
17:         m2=f_{n+1}, f_{n+2}, ..., f_k "}"
18:     end if
19:     return (m1, m2)
20: end procedure
```

Fig. 7. Intermediate file to PROMELA

4.4. Temporal Properties

Once a Stateflow model has been translated to PROMELA using either tool chain, temporal properties can be described in an external file for verifying desired behaviors. SPIN supports temporal operators such as <> (sometime), [] (always) and U (until). For example, the two key temporal properties considered in the examples below can be expressed as follows:

P1 **reachability:** [](*off* -><> (*active*))
P2 **liveness:** [](<> *active*)

Note that property P1 expresses that *active* can be reached from *off* by some sequence of transitions, i.e., **reachability**. Property P2 indicates that state *active* can eventually be reached starting from any other state, i.e., **liveness**. The interested reader should consult [12] for details on specifying temporal and other properties for checking using SPIN.

5. Experimental Results

5.1. Checking Reachability and Liveness

In this section, we show experimental results for the verification of a number of Simulink Stateflow models. Our experiments are performed on a 2.4 GHz Core i7 processor under OS X v10.8 with 8 GB of available RAM. All simulations are verified by SPIN version 6.2.7 and our tools are developed using Python 2.7.6.

Table 1 shows our results regarding reachability checking for a number of Simulink models from [5]. For each the table gives the name and the number of state variables. The section WSAT shows the results of applying the tool path employing WSAT and the section Our Tool shows the results of using our tool for direct translation to PROMELA. For each case, we show the number of states visited, the number of transitions taken and the total CPU time. The time value consists of `WSAT` or `PY` (depending on the tool chain), plus the time for `SPIN`, the time `gcc` for compiling the C code produced by SPIN and the run time `exec`. Note that the times quoted do not include the initial translation from XML to the intermediate format (Fig.3) or the translation from the intermediate format to MSL (Fig.5) to be input to the WSAT tool. Those times are relatively minor.

The reachability property as generated by SPIN is:

P1 [] ((*initial state*) -><>(*final state*(*s*))).

Table. 2 shows the results for liveness checking. In this case the SPIN property is:

P2 [] (<>(*final state*(*s*))).

In Table 1 and 2, all models are verified correctly for the properties of reachability and liveness.

Simulink model checking can only verify *deterministic* behavior which means that events occur in a particular order. For example, if a model can transition from $\{a\}$ to b or c with some probability, deterministic behavior requires either $\{a \rightarrow b \rightarrow c\}$ or

$\{a \rightarrow c \rightarrow b\}$. In this type of case were more than one transition is possible at the same time, one of them will be selected by the Simulink checker in a deterministic way using the Stateflow 12 o'clock rule[21]. In practical models, we have to consider true *nondeterministic* behavior which motivates our approach of translating the Simulink model to a form that can be checked nondeterministically using SPIN.

5.2. A Nondeterministic Example

In this section, we describe a nondeterministic Simulink Stateflow model as shown in Fig. 8. There are nondeterministic behaviors in *valid_state* and *failure_state*. Here, if the system has some nondeterministic behaviors, the results may cause failure operations. We use the PROMELA code shown in Fig.9 in order to check failure operations. In Fig.9, processes `valid_state` and `failure_state` can be expressed deterministically. Therefore, we can check nondeterministic behaviors of the Stateflow model by the SPIN model checker.

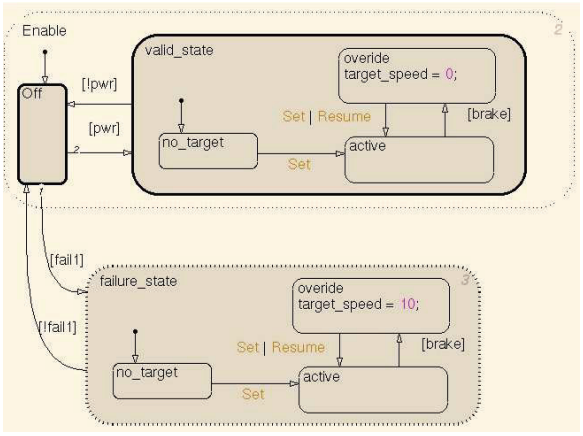

Fig. 8. Simulink Stateflow model (including failure)

```
active proctype nondeterm(){
mtype event;
mtype Enable, Off, pwr, fail1, Set, Resume, brake;

valid_state:
if
::(event==pwr) && (nondeterm_state==Off) ->
nondeterm_state=no_target
::(event==Set) && (nondeterm_state==no_target) ->
nondeterm_state=active
::(event==brake) && (nondeterm_state==active) ->
nondeterm_state=override
::else -> goto failure_state;
fi;

failure_state:
if
::(event==fail1) && (nondeterm_state==Off) ->
nondeterm_state=no_target
::(event==Set) && (nondeterm_state==no_target) ->
nondeterm_state=active
::(event==brake) && (nondeterm_state==active) ->
nondeterm_state=override
::else -> goto valid_state;
fi;
}
```

Fig. 9. PROMELA specification for nondeterministic behaviors.

Experimental results for nondeterministic models are shown in Tables 3 and 4. The examples in the top half of the table are from [22]. We created the examples in the bottom half of the table to illustrate the performance of our approach on larger examples. The client_server is extended from 2 to 12 nodes; the ring is extended from 3 to 36 nodes; and the ethernet example is extended from 4 to 22 nodes.

The results for reachability checking show that the number of states visited, the number of transitions applied and the CPU time used are are all reduced by using our new tool. Moreover, the results show especially efficient checking results for the large models. For liveness checking, the results show similar performance improvement.

It is interesting to note that the DTMF_generator example only has 8 state variables but requires visiting a large number of states and taking a large number of transitions. This is due to the particular structure of this example.

Our new approach consistently requires fewer state visits and transitions in testing reachability and liveness for both the deterministic and nondeterministic examples. Our method is also consistently faster and very much so for the large examples. Note that the main difference in time is between WSAT (Fig.6) and our tool in Python (Fig.7). Our tool is a much simpler approach resulting in a much faster

Table 1: Experimental results for checking reachability (deterministic)

		WSAT			Our Tool		
Simulink models	states	state visitss	trans.	time[s] (WSAT+SPIN+gcc+exec.)	state visitss	trans.	time[s] (PY+SPIN+gcc+exec.)
frame-sync-cont.	5	43	119	6.23 (1.48 + 1.24 + 3.24 + 0.27)	27	57	5.10 (0.31 + 1.26 + 3.28 + 0.25)
abs	6	51	137	6.75 (1.75 + 1.40 + 3.28 + 0.33)	19	44	4.76 (0.18 + 1.14 + 3.20 + 0.24)
dcmotor	8	24	55	6.01 (1.52 + 1.20 + 3.04 + 0.24)	9	10	4.75 (0.26 + 1.16 + 3.09 + 0.24)
aircontrol	9	35	80	6.31 (1.59 + 1.23 + 3.23 + 0.25)	32	73	5.11 (0.34 + 1.27 + 3.24 + 0.26)
ladder-logic	9	79	109	6.63 (1.73 + 1.33 + 3.32 + 0.25)	21	57	6.01 (0.28 + 1.38 + 4.11 + 0.25)
car	10	55	155	6.55 (1.82 + 1.24 + 3.25 + 0.25)	56	145	5.10 (0.30 + 1.23 + 3.33 + 0.25)
gfdemo	10	39	101	6.82 (1.91 + 1.30 + 3.36 + 0.25)	27	73	6.10 (0.27 + 1.32 + 4.25 + 0.26)
bridge	22	87	244	7.31 (2.45 + 1.26 + 3.34 + 0.26)	73	173	5.06 (0.30 + 1.23 + 3.29 + 0.25)
server	28	331	884	8.34 (3.41 + 1.25 + 3.43 + 0.25)	256	712	5.03 (0.37 + 1.16 + 3.25 + 0.25)
fuelsys-logic	34	75	170	7.89 (2.79 + 1.36 + 3.49 + 0.26)	51	116	5.20 (0.29 + 1.32 + 3.35 + 0.25)
aircraft	110	143	332	8.51 (2.81 + 1.52 + 3.94 + 0.25)	139	323	5.75 (0.54 + 1.46 + 3.49 + 0.25)

Table 2: Experimental results for checking liveness (deterministic)

		WSAT			Our Tool		
Simulink models	states	state visits	trans.	time[s] (WSAT+SPIN+gcc+exec.)	state visits	trans.	time[s] (PY+SPIN+gcc+exec.)
frame-sync-cont.	5	23	43	8.60 (1.48 + 1.22 + 5.59 + 0.32)	13	25	6.74 (0.31 + 1.51 + 4.60 + 0.32)
abs	6	27	49	7.83 (1.75 + 1.24 + 4.54 + 0.31)	11	18	6.31 (0.18 + 1.26 + 4.55 + 0.31)
dcmotor	8	15	24	7.67 (1.52 + 1.29 + 4.55 + 0.31)	12	19	6.53 (0.26 + 1.35 + 4.58 + 0.34)
aircontrol	9	19	30	6.28 (1.59 + 1.21 + 3.25 + 0.24)	16	25	5.07 (0.34 + 1.21 + 3.26 + 0.26)
ladder-logic	9	41	73	9.15 (1.73 + 1.53 + 5.57 + 0.32)	13	25	7.60 (0.28 + 2.28 + 4.72 + 0.32)
car	10	29	55	8.03 (1.82 + 1.36 + 4.54 + 0.31)	28	49	6.63 (0.30 + 1.31 + 4.72 + 0.31)
gfdemo	10	21	37	9.43 (1.91 + 1.52 + 5.68 + 0.32)	15	29	8.02 (0.27 + 2.63 + 4.79 + 0.32)
bridge	22	45	78	8.74 (2.45 + 1.32 + 4.66 + 0.31)	41	69	6.79 (0.30 + 1.32 + 4.86 + 0.31)
server	28	167	298	10.11 (3.41 + 1.38 + 5.00 + 0.32)	128	238	7.14 (0.37 + 1.33 + 5.13 + 0.32)
fuelsys-logic	34	59	80	11.87 (2.79 + 1.62 + 7.14 + 0.32)	47	62	8.26 (0.29 + 2.68 + 4.97 + 0.32)
aircraft	110	123	164	10.30 (2.81 + 1.55 + 5.63 + 0.31)	121	160	8.75 (0.54 + 1.67 + 6.22 + 0.31)

Table 3: Experimental results for checking reachability (nondeterministic)

		WSAT			Our tool		
Simulink models	states	state visits	trans.	time[s] (WSAT+SPIN+gcc+exec.)	state visits	trans.	time[s] (PLY+SPIN+gcc+exec.)
DTMF_generator	8	215	614	8.56 (3.01 + 1.56 + 3.76 + 0.23)	112	316	6.67 (0.46 + 2.06 + 3.92 + 0.23)
ring	12	51	128	8.06 (2.61 + 1.53 + 3.68 + 0.23)	48	118	6.00 (0.46 + 1.53 + 3.76 + 0.25)
clutch_model	18	99	251	8.14 (2.73 + 1.51 + 3.67 + 0.23)	97	243	6.24 (0.47 + 1.76 + 3.78 + 0.23)
client_server_bus	20	159	452	8.84 (3.29 + 1.53 + 3.79 + 0.23)	83	176	6.18 (0.53 + 1.56 + 3.83 + 0.27)
ethernet	32	79	197	13.02 (7.51 + 1.43 + 3.85 + 0.23)	71	161	6.19 (0.56 + 1.42 + 3.98 + 0.23)
packet_switch	32	147	386	15.21 (5.51 + 2.04 + 7.43 + 0.23)	120	325	6.65 (0.49 + 1.72 + 4.21 + 0.23)
callprocess	49	151	386	14.50 (8.10 + 1.79 + 4.35 + 0.25)	55	128	7.49 (0.75 + 2.03 + 4.48 + 0.23)
client_server_n12	100	507	1205	17.24 (10.36 + 1.72 + 4.88 + 0.28)	419	944	7.34 (0.55 + 1.74 + 4.77 + 0.28)
client_server_n12_err	100	411	989	16.52 (9.63 + 1.77 + 4.83 + 0.29)	335	755	7.18 (0.51 + 1.75 + 4.64 + 0.28)
ring_n36	145	611	1412	25.25 (17.36 + 1.93 + 5.68 + 0.28)	592	1342	8.67 (0.99 + 1.90 + 5.49 + 0.29)
ring_n36_err	145	323	764	22.75 (16.11 + 1.64 + 4.71 + 0.29)	304	694	7.60 (0.87 + 1.72 + 4.72 + 0.29)
ethernet_n22	176	283	656	21.31 (14.14 + 1.75 + 5.13 + 0.29)	193	441	8.02 (0.67 + 1.82 + 5.24 + 0.29)
ethernet_n23_err	176	247	575	19.60 (12.51 + 1.74 + 5.06 + 0.29)	159	359	8.00 (0.67 + 1.84 + 5.19 + 0.30)

Table 4: Experimental results for checking liveness (nondeterministic)

		WSAT			Our tool		
Simulink models	states	state visits	trans.	time[s] (WSAT+SPIN+gcc+exec.)	state visits	trans.	time[s] (PLY+SPIN+gcc+exec.)
DTMF_generator	8	109	208	8.45 (3.01 + 1.51 + 3.70 + 0.23)	56	106	6.00 (0.46 + 1.53 + 3.78 + 0.23)
ring	12	27	46	7.98 (2.61 + 1.51 + 3.62 + 0.23)	24	40	5.90 (0.46 + 1.51 + 3.71 + 0.22)
clutch_model	18	51	87	8.08 (2.73 + 1.51 + 3.60 + 0.23)	21	57	6.54 (0.47 + 1.51 + 4.32 + 0.23)
client_server_bus	20	81	154	8.75 (3.29 + 1.53 + 3.71 + 0.22)	52	76	6.07 (0.53 + 1.54 + 3.76 + 0.23)
ethernet	32	41	69	13.11 (7.51 + 1.49 + 3.88 + 0.23)	37	57	6.49 (0.56 + 1.41 + 4.29 + 0.23)
packet_switch	32	75	132	11.49 (5.51 + 1.75 + 4.00 + 0.23)	60	109	6.57 (0.49 + 1.67 + 4.18 + 0.23)
callprocess	49	77	132	14.85 (8.10 + 2.17 + 4.35 + 0.23)	53	82	7.44 (0.75 + 1.75 + 4.71 + 0.23)
client_server_n12	100	255	405	17.19 (10.36 + 1.69 + 4.85 + 0.29)	211	318	7.25 (0.55 + 1.70 + 4.71 + 0.29)
client_server_n12_err	100	207	333	16.25 (9.63 + 1.73 + 4.61 + 0.28)	169	255	7.14 (0.51 + 1.79 + 4.56 + 0.28)
ring_n36	145	307	474	25.12 (17.36 + 1.88 + 5.60 + 0.28)	296	448	8.59 (0.99 + 1.84 + 5.47 + 0.29)
ring_n36_err	145	163	258	22.68 (16.11 + 1.67 + 4.63 + 0.27)	152	232	7.46 (0.87 + 1.66 + 4.66 + 0.30)
ethernet_n22	176	243	422	21.73 (14.14 + 1.79 + 5.44 + 0.36)	183	413	7.95 (0.67 + 1.81 + 5.16 + 0.31)
ethernet_n23_err	176	125	195	19.53 (12.51 + 1.77 + 4.94 + 0.31)	81	123	7.91 (0.67 + 1.86 + 5.08 + 0.30)

translation process.

To better illustrate the application of the checking methods we have included results for variants (marked _err) of our large examples which contain errors that make the reachability and liveness tests fail. Note that the fail cases consistently require fewer state visits and transitions.

6. Conclusion

In this paper, we have described two tool chains that support using SPIN to model check systems specified as Simulink Stateflow models. This approach is based on existing commercial tools and techniques that are increasingly used for systems and software engineering for nondeterministic behaviors. We have developed tools for translating XML to MSL or PROMELA. We illustrated how these tools can translate and check important properties such as reachability and liveness. The approaches presented in this paper have several benefits to offer to next-generation nondeterministic behaviors analysis. In future work, we will work to embed our tool as a module in the Simulink Stateflow design flow, and consider developing a fully integrated tool for model checking.

Acknowledgment

This work was supported in part by a Grant-in-Aid for Creative Scientific Research No.40412902 of the Ministry of Education, Science, Sports and Culture (MEXT) of Japan and a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. We would like to thank Dr. Sudhakar Ganti for his technical support. We also acknowledge CMC Microsystems for the provision of products and services that facilitated this research.

References

1. J. E. M. Clarke, O. Grumberg, Peled, and D. A., *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
2. K. Nermin, P. Christopher, H. Andrea, and K. Christian, "On design-time modelling and verification of safety-critical component-based systems," *International Journal of Networked and Distributed Computing (IJNDC)*, vol. 2-3, pp. 175–188, July 2014.
3. H. Jun, W. Tao, G. Quan, and S. Gang, "Query integrity verification based-on mac chain in cloud storage," *International Journal of Networked and Distributed Computing (IJNDC)*, vol. 2-4, pp. 241–249, November 2014.
4. "Spin web site," http://www.spinroot.com.
5. "Simulink© documentation center (the mathworks, inc.)," http://www.mathworks.com/help/simulink/.
6. X. Fu, T. Bultan, and J. Su, "Wsat: A tool for formal analysis of web services." in *CAV*, ser. Lecture Notes in Computer Science, vol. 3114. Springer, 2004, pp. 510–514.
7. "Simulink© design verifier documentation center (the mathworks, inc.)," http://www.mathworks.com/help/sldv/.
8. F. Leitner, "Evaluation of the matlab simulink design verifier versus the model checker SPIN," Bachelor Thesis, University of Konstanz, 2008.

9. "Extensible markup language (xml)," http://www.w3.org/XML/.
10. A. Brown, M. Fuchs, J. Robie, and P. Wadler, "Msl — a model for w3c xml schema," in *Proceedings of the 10th International Conference on World Wide Web*, ser. WWW '01. New York, NY, USA: ACM, 2001, pp. 191–200.
11. X. Fu, T. Bultan, and J. Su, "Model checking xml manipulating software," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. ACM, 2004, pp. 252–262.
12. G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.
13. A. Agrawal, G. Simon, and G. Karsai, "Semantic translation of simulink/stateflow models to hybrid automata using graph transformations," *Electron. Notes Theor. Comput. Sci.*, vol. 109, pp. 43–56, Dec. 2004.
14. A. Joshi and M. P. E. Heimdahl, "Model-based safety analysis of simulink models using scade design verifier," in *Proceedings of the 24th International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP'05. Springer-Verlag, 2005, pp. 122–135.
15. N. Rouquette, J. Dunphy, and M. S. Feather, "A flexible statechart-to-model-checker translator," in *IEEE, International Symposium on Requirements Engineering*, 1999.
16. D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker," *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, 1999.
17. K. Jiang and B. Jonsson, "Using spin to model check concurrent algorithms, using a translation from c to promela," in *Proc. 2nd Swedish Workshop on Multi-Core Computing*. Department of Information Technology, Uppsala University, 2009, pp. 67–69.
18. P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán, "Model checking software with well-defined apis: The socket case," in *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, ser. FMICS '05. ACM, 2005, pp. 17–26.
19. P. de la Cámara, M. Gallardo, and P. Merino, "Abstract matching for software model checking," in *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, 2006, pp. 182–200.
20. "Ply (python lex-yacc)," http://www.dabeaz.com/ply/.
21. N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, "Defining and translating a "safe" subset of simulink/stateflow into lustre," in *Proceedings of the 4th ACM International Conference on Embedded Software*, ser. EMSOFT '04. New York, NY, USA: ACM, 2004, pp. 259–268.
22. "Matlab central," http://www.mathworks.com/matlabcentral/fileexchange/.