An Approach to Formal Verification of Embedded Software

MIROSLAV POPOVIC, ILIJA BASICEVIC
Faculty of Technical Sciences
University of Novi Sad
Trg Dositeja Obradovica 6, 21000 Novi Sad
SERBIA
miroslav.popovic@rt-rk.com    http://www.ftn.uns.ac.rs

Abstract: - Modern distributed large-scale systems comprise very large number of embedded processors, which are running embedded software. The complexity of these systems of systems is so high that it becomes unmanageable by humans. Under such circumstances, formal methods and the corresponding tools is being subject of intensive research and development in both industry and academia. The objective of this paper is to make a contribution to the overall efforts by proposing a method, and accompanying tools, for formal verification of a class of embedded software that may be modeled as a collection of distributed finite state machines. The method is based on the symbolic model verification of certain properties of embedded software models. The accompanying tools enable creation of these models from the high-level design models and/or from the target program code, e.g. in C/C++ language. The viability of the proposed method is demonstrated on a case study. The subject of the case study is the verification of distributed embedded software that executes in the telephone switches and call centers. The results of the case study show that the proposed method is applicable on the real-world systems.

Key-Words: - Embedded software, Formal verification, Telephone exchange, Call center, SDL language, SMV language, Model checking, Software tools

1 Introduction

Nowadays embedded software is used everywhere. It is present in the whole spectrum of systems, starting from relatively simple hand held devices, across home appliances, vehicles, and ending with the large-scale systems, such as air plains, telephone network, Internet, electricity power distribution networks, etc. Modern distributed large-scale systems comprise very large number of embedded processors, which are running embedded software. The complexity of these systems of systems is so high that it becomes unmanageable by humans.

Under such circumstances, formal methods and the corresponding tools are being subject of intensive research and development in both industry and academia. Although formal methods are being successfully applied in hardware design of conventional CUPs, new multicore processors, multiprocessor computers, and parallel architectures are opening new challenges for their formal verification. On the other hand, even though there are some promising results of formal verification of embedded software in the area of mission-critical infrastructure, it is far from being considered as a routine practice in the mainstream commercial industry. The objective of this paper is to make a contribution to the overall efforts in this area by proposing a method, and accompanying tools, for the formal verification of a class of embedded software that may be modeled as a collection of distributed finite state machines.

The method is based on the symbolic model verification of certain properties of embedded software models expressed in the SMV language [1]. The proposed method specifies a systematic procedure that can be used to create the software model and its properties from the given embedded software specification, e.g. in ITU-T SDL language, and the given test suite defined for verifying software compliance to the specification. The resulting set of model properties may be extended manually with the additional ad-hoc model properties based on the intuition and experience of engineers doing the verification.

The accompanying tools enable creation of these models from the high-level design models and/or from the target program code, e.g. in C/C++ language. The prototypes of these tools are based on the previously developed tools [2], [3] and [4], and they are under development as of the time of these writings. The viability of the proposed method is demonstrated on a case study. The subject of the case study is the verification of distributed embedded software that executes in the telephone switches and call centers. More details about the latter may be found in [5] and [6]. The results of the case study show that the proposed method is applicable for the real-world systems. We hope that this paper may
inspire other researchers to develop similar methods and tools. We also hope that practitioners will find useful the approach presented in this paper and that it will help them to manage their own projects.

The text of the paper is organized as follows. The related work is presented in the next subsection. Modeling of the target class of embedded software and the proposed method are covered in Section 2 and 3, respectively. The case study is presented in Section 4. The final conclusions are given in Section 5.

1.1 Related work
This subsection provides a brief coverage of the state of the art methods and tools for the embedded software verification [7-10].

Generally, model checkers are formal verification tools that evaluate a model to determine if it satisfies a given set of properties, see [7]. Modern symbolic model checkers use logical representations of sets of states, such as BDDs (Binary Decision Diagrams), to represent regions of the state space, which satisfy the properties being evaluated. For example, a BDD-based model checker that we used in this paper [1] can effectively analyze models with over $10^{100}$ reachable states. Furthermore, model checkers like SAL and Prover Plug-In use SMT (Satisfiability Modulo Theories) to analyze infinite state models. Although most embedded software nowadays is still modeled as FSMs (Finite State Machines), emerging SMT based model checkers enable model checking of future ISM (Infinite State Machine) models. Finally, practitioners may use translators to combine popular modeling languages and various model checkers and theorem provers, e.g. see [8].

The results provided by these modeling languages and tools are promising, but there are still open issues. For example, lessons learned from the three case studies [9], related to the verification of embedded software in the aircraft industry, indicate that determining what properties to verify may be a difficult problem. It can also be difficult to determine how many properties must be checked. Their experience is that checking even a few properties will find errors, but that checking more properties will find more errors. In this paper we are addressing this issue, and we are making a contribution by proposing a method that systematically generates the properties that should be verified for a class of embedded software that may be modeled as a collection of distributed FSMs. These properties are generated by translating the given test suite originally used for compliance testing into the SMV properties.

David Parnas in his recent and provoking paper on rethinking formal methods [10] discusses a list of open issues in a form of open questions to the community. His general message is that those issues should be revisited and perhaps approached in a different way than they are treated today. One of those issues is a question: should time be treated as a special variable or just another variable? Historically, special logics were developed for dealing with time issues. This is quite different from control theory and circuit theory, where time is represented by an additional variable that is not treated in any special way. Parnas concludes that rethinking would require serious consideration of this alternative. We agree, and in this paper we show how to replace a set of timers that are managed by an individual FSM with an enumerated variable, which represents the time.

2 Modeling
In this section we present an approach to model a collection of distributed FSMs, such as communication protocols, in SMV language. The communication protocols are typically specified in the ITU-T SDL language, UML state-charts, UML activity diagrams, or classical state transition graphs as the ones used in hardware design. This section describes how to encode any collection of such FSMs in SMV language, and it does not depend on the language that is used in the original specification of a collection of FSMs. The first subsection describes modeling of individual FSMs, whereas the second subsection covers modeling a collection of cooperating FSMs, which may be deployed on geographically distributed machines.

2.1 Modeling Individual FSMs
A FSM is modeled as a module with the given name and a list of input and output parameters:

\[
\text{module } \text{name} \ (\text{param}_1 \ , \ \text{param}_2 \ \ldots \ \text{param}_o1 \ , \ \text{param}_o2 \ \ldots) \ \{ \\
\ldots \\
\}
\]

All the possible values of all the parameters are enumerated inside the module definition:

\[
\begin{align*}
\text{input } \text{param}_i1 & : \{ \\
\text{none} , \text{param}_i1\_value_1 , \\
\text{param}_i1\_value_2 , \ldots \\
\} \\
\ldots \\
\text{output } \text{param}_o1 & : \{ \\
\text{none} , \text{param}_o1\_value_1 , \\
\text{param}_o1\_value_2 , \ldots \\
\} \\
\ldots \\
\end{align*}
\]

Here, individual parameter values correspond to particular messages exchanged by FSMs. The value \text{none} is a special value that represents the absence of any meaningful message. This value corresponds to the three-state signal in the area of hardware design.
Then, all the possible FSM states are enumerated as the possible values of the state variable, which is assigned the initial value that corresponds to the initial state:

```plaintext
state : {
    STATE_1, STATE_2, ...
}
init(state) := STATE_1;
```

Also, if the FSM maintains any timers, the corresponding timer expiry moments are enumerated as the possible values of the variable time:

```plaintext
time : {
    t0, T1, T2, ...
}
```

The value $t0$ represents the FSM operation starting time. The value $T1$ corresponds to the moment when the first timer expires, the value $T2$ corresponds to the moment when the second timer expires, and so on.

If the FSM has any additional state variables and/or operational variables, e.g. dependant on the values of message parameters, they are also declared and initialized accordingly. After all the declarations and initializations are made, the behavior of the FSM is defined as a series of else-if clauses, which of course starts with the initial if clause:

```plaintext
if(precondition_1)
    {action_1}
else if(precondition_2)
    {action_2}
...
```

This particular definition of the FSM behavior was selected because it can be easily generated from the output of the tool described in [3]. The preconditions in the FSM behavior definition are the conjunctions of the equalities on the state variables and the input parameters, or the state variables and the variable time.

The actions are the lists of the assignments that are assigning the next values to the state variables, the output parameters, and/or the variable time.

Each if or else-if clause defines a FSM reaction to a given event (e.g. reception of a message or expiry of a given timer). A FSM reaction is typically fired be the reception of a given message on its input, and as the result of the reaction, FSM moves to the new state and generates a corresponding message on its output. For example, the following if clause defines that if the FSM is in the state FE2_IDLE and it receives the message r1_SetupReqInd, it will make a transition into the state FE2_WAIT_FOR_DIGITS and it will send the message r1_ProceedingReqInd:

```plaintext
if(state=FE2_IDLE&fin1=r1_SetupReqInd)
    {next(state) := FE2_WAIT_FOR_DIGITS;
     next(fout1) := r1_ProceedingReqInd;}
```

2.2 Modeling Collections of FSMs

A collection of FSMs is modeled in a separate module. In the case when the system has just one collection of FSMs it may be modeled in the module main. Multiple collections of FSMs may be used to model different subsystems located on the same node or on the different nodes of the network. At the beginning of the module all the variables that are used to interconnect communicating FSM are declared. Then all the FSMs in the collection are instantiated. The FSMs are interconnected by an appropriate arrangement of the input/output parameters of individual FSMs, and when needed by additional assignments of the output parameters to the input parameters.

For example, in the simple case when two FSMs communicate to each other over dedicated input and output parameters, it is sufficient to declare two variables. The first variable is used as the output parameter of the first FSM and the input parameter of the second FSM, whereas the second variable is used as the output parameter of the second FSM and the input parameter of the first FSM:

```plaintext
v1 : {...};
v2 : {...};
fsm1_instance : fsm1(v1,v2);
fsm2_instance : fsm2(v2,v1);
```

For the case when outputs of more than one FSM have to be connected together to the same input of some FSM, the associated input parameter is assigned the result of the union of the associated output parameters. For example, consider the case when the outputs of the FSM1 and the FSM2 are connected to the input of the FSM3:

```plaintext
v1 : {...};
v2 : {...};
v3 : {...};
fsm1_instance : fsm1(...,v1);
fsm2_instance : fsm2(...,v2);
fsm3_instance : fsm2(v3,...);
v3 := v1 union v2;
```

3 Method

The method, which is used in this paper for the formal verification of a class of embedded software, is based on the modeling approach presented in the previous section. The proposed method specifies a systematic procedure that can be used to create the SMV model and the model properties from the given embedded software specification and the given test suite, respectively. The
The given test suite is normally used for verifying software conformance to the specification.

The resulting set of model properties may be extended manually with the additional ad-hoc model properties. The SMV model checker effectively verifies the software by checking the resulting model properties of the resulting SMV model. The method comprises the following steps:

**Step1:** The high-level embedded software specification (a.k.a. software model) in form of ITU-T SDL (Specification and Description Language) diagrams is entered into the SDL editor. The SDL editor is typically a part of IDE (Integrated Development Environment), such as the one described in the paper [2]. This step is optional, but most usually it is performed. The step may be skipped if the high-level software specification or the tools are not available. If this step is skipped, then SDL diagrams have to be manually transformed into the target program code in the next step.

**Step2:** The SDL diagrams are automatically translated by the SDL compiler into the target program code. For example, the target code may be the C++ code, which runs on top of the FSM library. The FSM library [4] is the specific run-time library that supports applications based on the distributed groups of FSMs. Alternatively the SDL diagrams may be coded manually. For example, they may be coded in C++ using the restricted programming paradigm enforced by the FSM library.

**Step3:** The axiomatic specification of individual FSMs is automatically extracted from the target program code. For example, the axiomatic specification may be extracted from the target code by the reverse engineering tool, as the one described in [3]. The resulting axiomatic specification is translated into the SMV model by the tool that we are currently developing to aid this translation.

**Step4:** The test suite normally used for the implementation conformance testing, is translated into a set of corresponding theorems, which are in turn translated to the corresponding SMV model properties. Most commonly a test suite would be given in the ITU-T TTCN (Tree and Tabular Combined Notation) language. Optionally, additional application specific model properties may be written manually to assert general properties, such as safety and liveness that are typically provided through the mutual exclusion and services, as well as the absence of race conditions, deadlocks, and live-locks.

**Step5:** The model properties are automatically checked by the SMV model checker [1]. Typically, manually written model properties are added incrementally in iterations in which the previous step number 4 and this step are repeated. The properties are added incrementally because defining ad hoc model properties depends on the human intuition and experience. While working on verification of particular software, humans get better insight in the inner workings of that particular software. Sometimes they succeed to define more general properties after defining some less general properties in the beginning. Sometimes they make mistakes by defining asserts that are not valid properties of that particular software. That is why this process is incremental in its nature.

We may summarize the procedure as follows. The target program code, e.g. in C++, is either manually written (if the step 1 above is skipped), or automatically generated (if the step 1 is not skipped). SMV model is then extracted from the target program code using appropriate reverse engineering tools. That is exactly the first essential idea of the proposed method: the high-level software specification is not translated to the SMV model; rather it is reverse engineered from the target program code that we want to verify. On the other hand, the initial set of SMV model properties is derived from the given test suite that is used for software acceptance testing, which leads us to the second essential idea of the proposed method: the initial set of model properties is systematically derived from the given test suite, rather then being ad hoc written based on human intuition.

As already mentioned in the subsection on related work, selecting properties to verify may be a difficult problem. This proves to be true for the class of embedded software analyzed in this paper. Therefore we propose to rely on the given test suite and to verify at minimum the model properties derived from the given test suite. The tests from the test suite that is used for the implementation conformance testing are normally divided into the six categories [11].

## 4 Case Study

In this section we demonstrate the applicability of the proposed method by means of a case study. The subject of the case study is the formal verification of the local call processing software, which has been implemented in accordance with the ITU-T Q.71 recommendation. The local call processing software comprises four FSMs, which are referred to as FEs (Functional Entities) in the Q.71 recommendation. The overall functionality of the local call processing software is to establish and release of calls between the calling and called party, which are typically referred to as user $A$ and user $B$, respectively. The architecture of the resulting SMV model is illustrated in the Fig. 1. Despite the fact that this study may seem too simple, it actually captures all the significant aspects of the existing software in telephone exchanges, call centers, and similar communications systems.
As shown on the left hand side of the Fig. 1, the FSMs User_A and User_B, model the calling and the called user, respectively. The remaining FSMs that are shown in the Fig. 1 model the FEs according to ITU-T Q.71 recommendation. As previously mentioned in the section on moulding collections of FSMs, FSMs communicate through the input and the output variables. These variables create communication channels that are shown in the Fig. 1 as the links connecting the corresponding FSMs. For example, the FSM User_A communicates directly only with the FSM FE1, the FSM FE1 communicates directly with the FSM User_A and with the FSM FE2, and so on. Obviously, the FSMs User_A and User_A communicate indirectly over a chain of four FEs, namely the FE1, FE2, FE3, and FE4.

The data about the size of the SMV model that models the collection of FSMs shown in the Fig. 1 is given in the Tab. 1. The rows of the Tab. 1 show data about the individual FSM, whereas the columns of the Tab. 1 indicate the number of states, the number of distinct input messages, the number of distinct output messages, the number of timers, and the number of lines of SMV code.

We derive individual model properties from individual test cases of the given test suite. A test case comprises a series of test steps. Each test step is triggered by an event, typically a FSM receives a message in some of its states, and the test step results in a certain action, typically the FSM sends a message, and transits to the next state. These test steps are encoded as individual implications, and a complete test case is encoded as a series of properly parenthesized implications so that they are evaluated from left to right.

Next we present the two typical model properties as examples of the properties that were derived from the conformance test suite and successfully checked by the SMV model checker. The two sample model properties are the following:

1. The successful call establishment (SCE): the user A initiates the call by sending the hook-off signal, dials a number (a single digit in this example), and the user B accepts the call by sending the hook-off signal.

2. The expiry of the inter-digit timeout (IDT): the user A initiates the call by sending the hook-off signal, then it fails to send the digit, therefore timer T1 maintained by the FSM FE2 expires (user receives the busy tone), and finally user A disconnects by sending the hook-on signal.

The SCE model property is the following:

```plaintext
prt_SCE: assert F
((fe1_in=User_OFF_HOOK ->
  s_fe1=FE1_UNKNOWN_FE2 &
  fe1_out=r1_SetupReqInd) ->
  (fe1_in=User_DIGIT & fe2_t<FE2_T1 ->
    s_fe1=FE1_UNKNOWN_FE2 &
    s_fe2=FE2_CALL_SENT &
    s_fe4=FE4_CALL_SENT &
    s_fe5=FE5_WAIT_OFF_HOOK)) ->
  (fe5_in=User_OFF_HOOK ->
    s_fe1=FE1_ACTIV & s_fe2=FE2_ACTIV &
    s_fe4=FE4_ACTIV & s_fe5=FE5_ACTIV);
```

We read the SCE model property as follows. When the user A sends the off-hook signal User_OFF_HOOK, the FSM FE1 will transit into the state FE1_UNKNOWN_FE2 and send the signal r1_SetupReqInd. Then if the user A sends the signal User_DIGIT and the time variable fe2_t is less than FE2_T1 (which means that the timer T1 maintained by FE2 is still running), FE2 transits into the state FE2_CALL_SENT, FE4 transits into the state FE4_CALL_SENT, and FE5 transits into the state FE5_WAIT_OFF_HOOK. At the end, when the user B answers the call, by sending the signal User_OFF_HOOK, all the FEs transit into the active state, which means that the call is successfully established.

The IDT model property is the following:

```plaintext
prt_IDT: assert F
  ((fe1_in=User_OFF_HOOK ->
    s_fe1=FE1_UNKNOWN_FE2 &
    fe1_out=r1_SetupReqInd) ->
    (fe2_t=FE2_T1 ->
      s_fe2=FE2_DISCONNECTING_FE1)) ->
    (fe1_in=User_ON_HOOK ->
      s_fe1=FE1_ON_HOOK & s_fe2=FE2_IDLE &
      s_fe4=FE4_IDLE & s_fe5=FE5_ON_HOOK);
```

---

**Table 1. The size of the SMV model**

<table>
<thead>
<tr>
<th>FSM</th>
<th>No of states</th>
<th>No of inputs</th>
<th>No of outputs</th>
<th>No of timers</th>
<th>No of lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE1</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>68</td>
</tr>
<tr>
<td>FE2</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>103</td>
</tr>
<tr>
<td>FE4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>95</td>
</tr>
<tr>
<td>FE5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>50</td>
</tr>
</tbody>
</table>
This property is interpreted as follows. At the beginning the user A sends the off-hook signal and FE1 in its turn transits into the state FE1_UNKNOWN_FE2 and sends the signal r1_SetupReqInd. Then the user A does not send the digit and the timer T1 (maintained by FE2) expires (subscriber A receives the busy tone). At the end, the user A disconnects by sending the on-hook signal.

The Tab. 2 shows the reachable states and the property checking times for the previous properties.

Table 2. The reachable states and property checking times [sec]

<table>
<thead>
<tr>
<th>Property</th>
<th>Reachable states</th>
<th>User time</th>
<th>Sys time</th>
<th>Model time</th>
</tr>
</thead>
<tbody>
<tr>
<td>prt_SCE</td>
<td>13291</td>
<td>0.125</td>
<td>0.015</td>
<td>0.031</td>
</tr>
<tr>
<td>prt_SCR</td>
<td>38494</td>
<td>0.218</td>
<td>0.015</td>
<td>0.062</td>
</tr>
<tr>
<td>prt_IDT</td>
<td>10331</td>
<td>0.031</td>
<td>0.015</td>
<td>0.031</td>
</tr>
</tbody>
</table>

Finally, it seems appropriate to mention that two program logic errors were discovered during this case study, which were not discovered by the previously conducted testing.

5 Conclusion

In this paper we proposed a method, with accompanying tools, for formal verification of a class of embedded software that may be modeled as a collection of FSMs. We addressed two open issues that were identified in the recent literature [9] and [10].

Firstly, the authors of [9] indicated that determining what properties to verify may be a difficult problem. One of the causes that make this difficult is the fact that specifying model properties is still predominantly an ad hoc process. Therefore, in this paper we propose a systematic method of specifying model properties by translating the given conformance test suite, typically given in the ITU-T TTCN language, for a class of embedded software we were dealing with in this paper.

Secondly, we treat time as just another enumerated variable whose values are periods of timers maintained by the FSM, as independently suggested by the author of [10]. This approach provides more expressive statements of model properties when related to time, because it explicitly shows the value of time in seconds. For example we may write the expression \( time < T1 \) to specify that current time is less than \( T1 \), which means that the timer \( T1 \) is still running. Or for example we may write the expression \( T1 \leq time < T2 \) to specify that the current time is greater or equal to \( T1 \) and less than \( T2 \), which means that timer \( T1 \) has expired, but that the timer \( T2 \) did not. Traditional representation of timeouts, in form of events that are typically encoded as special messages, is less expressive, because it does not explicitly show the value of time.

The usability of the proposed method has been successfully demonstrated by the means of a case study, the verification of the real call processing embedded software that runs in the existing telephone switches and call centers. Other researchers may be motivated by this paper to explore similar methods to systematically generate model properties and to introduce more expressive models, whereas the practitioners may find useful the approach to modeling collections of FSMs, as well as the concepts related to accompanying tools.

Acknowledgments

This work has been partly supported by the Serbian Ministry of Science, through grants III 44009 and TR 32031, 2011-2014.

References: