Formal verification of embedded software based on software compliance properties and explicit use of time

Miroslav Popovic and Ilija Basicevic

***Abstract*—**The complexity of embedded software running in modern distributed large-scale systems is going so high that it becomes hardly manageable by humans. Formal methods and the supporting tools are offering effective means for mastering complexity, and therefore they are remaining to be an important subject of intensive research and development in both industry and academia. This paper makes a contribution to the overall R&D efforts in the area by proposing a method, and supporting tools, for formal verification of a class of embedded software, which may be modeled as a collection of distributed finite state machines. The method is based on the model checking of certain properties of embedded software models by Cadence SMV tool. These properties are systematically derived from the compliance test suites normally defined by relevant standards for compliance software testing, and therefore we refer to them as the compliance software properties. Another specificity of our approach is that we enable explicit usage of time within the software properties being verified, which gives more expressiveness to these properties and bring them more close to system properties that are analyzed in other engineering disciplines. The supporting tools enable generation of these models from the high-level design models and/or from the target source code, for example in C/C++ language. We demonstrate the usability of the proposed method on a case study. The subject of the case study is formal verification of distributed embedded software actually used in real telephone switches and call centers.

***Keywords*—**Embedded software, Formal verification, Model checking, SDL language, SMV language.

I. 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 planes, 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

Manuscript received ___: Revised version received ___. This work was supported in part by the Serbian Ministry of Science and Technology Development under Grants III-44009 and TR-32031, 2011-2014.

M. P. is with the Faculty of Technical Sciences, University of Novi Sad, 21000 Novi Sad, Serbia (phone: +381-21-4801-101; fax: +381-21-450-721; e-mail: miroslav.popovic@rt-rk.com).

I. B. is with the Faculty of Technical Sciences, University of Novi Sad, 21000 Novi Sad, Serbia (e-mail: ilija.basicevic@rt-rk.com).

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.

A. *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.

Aoki and Matsuura recently proposed a method, based on model checking, for detecting hard-to-discover defects in enterprise systems [14]. Their approach is very similar to work of Achenbach and Ostermann [15]. In their approach, Aoki and Matsuura manually rewrite ABAP programs into Java programs, and then automatically translate Java programs into appropriate UPPAAL abstracts models, which they use to observe the program behavior. Their method of specializing models and defining model properties in the course of detecting defects heavily depends on the inspectors familiarity with the basic design of the target program and on their intuition, whereas our approach relies on the existing international standards, as will be shown shortly in the following text. Also, although they use UPPAAL language, they do not use the explicit notion of time in their model properties in the presented case study, whereas we do.

Kum *et al*. proposed a design methodology for safety model in automotive software architecture [16], which is based on AUTOSAR [17]. In their paper [16], Kum *et al*. presented their Context Action Reaction (CAR) logic, which they use to reason about the vehicle environment, driver actions, and vehicle reactions. Similarly to our approach, they also use the three cooperating FSMs to model the environment, the driver actions, and the vehicle reactions. As means of illustration of their approach, they prove two ad hoc defined safety properties using sequent calculus. However, they do not provide a systematic method of defining system properties to be check like we do. They also do not use the explicit notion of time in their model properties; rather they use it implicitly as an argument of predicates in their logic, e.g. they use the expression *travel*(*t*-4, *t*) to model the four hours of travel.

Yamada, Nakaga, and Nakahodo proposed the check-points extraction method by which temporal formulas can be obtained inductively from a given system specification, based on the notion of strong and weak temporal relations [18]. They use this method to reduce a system signal transition graph and therefore reduce the size of the model that is the subject of model checking. Advantage of their approach is seen in smaller OBDD size when compared to traditional approach on some arbitration modules, which they model checked by NuSMV tool. Although this a useful method to reduce the OBDD size, it does not foresee explicit usage of time in model properties that are to be checked.

Pura, Patriciu, and Bica presented how AVISPA formal verification tool can be used to validate the security properties of implicit on-demand ad hoc secure routing protocols. In order to prove the technique, they demonstrated it in a case study: formal verification of the ARAN protocol. However, they pointed out that since AVISPA does not support time, they were only able to prove the weak authentication security goal, and they were not able to check the standard

authentication prescribed by the ARAN protocol.

Millan *et al*. introduced an OCL extension for checking and transforming UML models [20]. The extended OCL language, which they call pOCL (procedural OCL), has two parts – one for the simultaneous access and manipulation of several models and the other for the introduction of transformation primitives. Although this approach is not directly related to model checking, it would be interesting to see, would it be possible to make a use of a language, such as pOCL, to check model properties typically checked by a model checker. Another interesting application of their approach would be to convert UML models, or SDL models, into SMV models. These are the applications we plan to study in more detail in our future work, and we believe these are interesting to broader research community.

II. 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.

A. Modeling Individual FSMs

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

```
module name (param_i1, param_i2 ...
             param_o1, param_o2 ...) {
  ...
}
```

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

```
  input param_i1 : {
    none, param_i1_value_1,
    param_i1_value_2, ...
  }
  ...
  output param_o1 : {
    none, param_o1_value_1,
    param_o1_value_2, ...
  }
  ...
```

Here, individual parameter values correspond to particular messages exchanged by FSMs. The value *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:

```
  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*:

```
  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:

```
  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*:

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

B. 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:

```
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:

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

III. 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 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 following six categories:

1. Basic interconnection tests check if there is a sufficient conformance for the interconnection of communicating FSMs, and are the selected parameters valid for the given configuration.
2. Capability tests check whether the declared capabilities are observable.
3. Valid behavior tests check the message sequence and the message contents.
4. Inopportune behavior tests check proper behavior when implementations are exposed to invalid sequences of valid messages.
5. Invalid behavior tests check proper behavior when implementations are exposed to invalid messages.
6. Timer expiry and counter mismatch tests check proper reactions to timer expiries and counter mismatches.

Detailed description of these six test categories may be found in the related ISO recommendation [11], and it goes outside the scope of this paper.

IV. Case Study

In this section we demonstrate the applicability of the proposed method by means of a case study (a shorter version of this case study is presented in [12]). 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.

Fig. 1 The interconnection of FSMs for the local call processing

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 modelling 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 Table 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.

Table 1. The size of the SMV model

FSM	No of states	No of inputs	No of outputs	No of timers	No of lines
FE1	6	7	5	0	68
FE2	7	6	8	2	103
FE4	4	6	6	3	95
FE5	4	5	4	0	50

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 three 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 three 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 successful call release (SCR): both user *A* and *B* disconnect by sending the hook-on signal.
3. 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:

```
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 complete message sequence during the call establishment is illustrated in Fig. 2.

The SCR model property is the following:

```
prt_SCR: assert F
   (s_fe1=FE1_ACTIV & s_fe2=FE2_ACTIV &
    s_fe4=FE4_ACTIV & s_fe5=FE5_ACTIV) ->
   (fe1_in=User_ON_HOOK &
    fe5_in=User_ON_HOOK ->
    s_fe1=FE1_ON_HOOK & s_fe2=FE2_IDLE &
    s_fe4=FE4_IDLE & s_fe5=FE5_ON_HOOK);
```

This property asserts that when all the FEs are in their active states, and both users disconnect by sending the signal *User_ON_HOOK*, all the FEs will finally transit to their inactive (on-hook and idle) states, which means that the call is successfully released. The complete message sequence during the call release is illustrated in Fig. 3.

The IDT model property is the following:

```
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);
```

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

Table 2. The reachable states and property checking times

Property	Reachable states	User time [s]	Sys time [s]	Model checking time [s]
prt_SCE	13291	0.125	0.015	0.031
prt_SCR	38494	0.218	0.015	0.062
prt_IDT	10331	0.031	0.015	0.031

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.

Finally, it is worth mentioning that recently we used the same formalism in another case study [13] to formally verify a distributed transaction management solution in a SOA based control system. The results of [13] show that the formalism presented in this paper may be successfully used for formal verification of critical system aspects, related to complex, and therefore hard to follow logic, commonly found within large-scale distributed systems, such as SOA.

V. 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. The method is based on the symbolic model verification of SMV models, which are automatically created from the target program code, e.g. in C/C++ code, which in turn may be created from the high-level design models, e.g. in ITU-T SDL language. The foundation for the method is the proposed approach to modeling individual FSMs and collections of FSMs. While presenting our approach to modeling of FSMs and the proposed method, we addressed two open issues that were identified in the recent literature, see [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. This paper motivates future activities of both researchers and practitioners. The former may find it inspiring to explore and invent similar methods to systematically generate model properties and to introduce more expressive models, whereas the latter may find useful the approach to modeling collections of FSMs, as well as the concepts related to accompanying tools.

REFERENCES

[1] K.L. McMillan, "The SMV language", Cadence Berkeley Labs, 1999, pp. 1-49.
[2] I. Velikic, M. Popovic, and V. Kovacevic, "A Concept of an Integrated Development Environment for Reactive Systems", *Proc. of IEEE ECBS*, 2004, pp. 233-240.
[3] M. Popovic, V. Kovacevic, and I. Velikic, "A Formal Software Verification Concept Based on Automated Theorem Proving and Reverse Engineering", *Proc. of IEEE ECBS*, 2002, pp. 59-66.
[4] M. Popovic, *Communication Protocol Engineering*, CRC Press, Boca Raton, FL, USA, 2006, ch. 5.
[5] M. Popovic, B. Atlagic, and V. Kovacevic, "Case study: a maintenance practice used with real-time telecommunication software", *Journal of Software Maintenance and Evolution Research and Practice*, John Wiley & Sons, Ltd., No. 13, pp. 97-126, 2001.
[6] M. Popovic and V. Kovacevic, "An Approach to Internet-Based Virtual Call Center Implementation", Networking - ICN 2001, Part I, *LNCS*, Vol. 2093/2001, P. Lorenz, Ed., Springer, 2001, pp. 75-84.
[7] E. Clarke, O. Grumberg, and D. Peled , *Model Checking*, The MIT Press, Cambridge, MA, 2001.
[8] S. Miller, A. Trible, and M. Whalen, M.P.E. Heimdahal, "Proving the Shalls", *International Journal on Software Tools for Technology Transfer*, Feb. 2006.
[9] S. Miller and M. Whalen, D. Cofer, Software Model Checking Takes Off, *Comm. of ACM*, Vol. 53, No. 2, 2010, pp. 58-64.
[10] D. Parnas, "Really Rethinking Formal Methods", *Computer*, Jan. 2010, pp. 28-34.
[11] TTCN notation for validation and verification, ISO/IEC 9646 (X.290).
[12] M. Popovic and I. Basicevic, "An Approach to Formal Verification of Embedded Software", *Proc. of 15th WSEAS Int. Conf. on COMPUTERS*, 2011, to be published.
[13] I. Popovic, V. Vrtunski, and M. Popovic, "Formal verification of Distributed Transaction Management in a SOA Based Control System", *Proc. of IEEE ECBS*, 2011, pp. 206-215.
[14] Y. Aoki and S. Matsuura, "A method for Detecting Unusual Defects in Enterprise System Using Model Checking Techniques", *Proc. of 10th WSEAS Int. Conf. SEPADS*, 2011, pp. 165-171.
[15] M. Achenbach and K. Ostermann, "Engineering Abstractions in Model Checking and Testing", *Proc. of 9th IEEE Int. Working Conf. SCAM*, 2009, pp. 137-146.
[16] S. Chandrasekaran, R. P. Vijaya, and R. S. Vijayravikumaran, "CAR Based Safety Model in Automotive Software Engineering", *Proc. of 10th WSEAS Int. Conf. SEPADS*, 2011, pp. 206-201.
[17] D. Kum, G. M. Park, S. Lee, and W. Jung, "AUTOSAR Migration from existing Automotive Software", *Proc. of Int. Conf. ICCAS*, 2008, pp. 558-562.
[18] C. Yamada, Y. Nakaga, and M. Nakahodo, "An Efficient Model Checking Using Check-Points Extraction Method" *Int. J. of Computers*, vol. 1, no. 3, pp. 95–101, 2007.
[19] M. L. Pura, V. V. Patriciu, and I. Bica, "Modeling and formal verification of implicit on-demand secure ad hoc routing protocols in HLPSL and AVISPA" *Int. J. of Computers and Communications*, vol. 3, no. 2, pp. 25–32, 2009.
[20] T. Millan, L. Sabatier, T. T. Le Thi, P. Bazex, and C. Percebios, "An OCL extension for checking and transforming UML models", *Proc. of 8th WSEAS Int. Conf. SEPADS*, 2009, pp. 144-149.

Miroslav V. Popovic was born in Novi Sad, Serbia on February 1, 1961. He received his M.Sc. degree in electrical engineering from the Faculty of technical sciences at the University of Novi Sad, Novi Sad, Serbia, in 1984, and his Ph.D. degree in electrical and computer engineering from the University of Novi Sad, Novi Sad, Serbia, in 1990. His major field of study was computer engineering.

He started his career as an assistant professor at the Faculty of technical sciences, where he remained working to the present day. He was promoted to a lecturer (docent) in 1992 and to an associated professor in 1997. Finally, he was promoted to a tenured professor in 2002. He is currently the head of the Chair of computer engineering and can be reached at the University of Novi Sad, Faculty of technical sciences, Department of computing and control, Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia. He wrote the book Communication Protocol Engineering (Boca Raton, Florida, USA: CRC Press, 2006) and about 150 papers published in international and domestic journals and conference proceedings. His current research interests are in the area of Engineering of computer based systems (ECBS), especially model-based development, test, and verification.

Prof. Popovic is the member of the program committee of the IEEE Annual Conference on Engineering of Computer Based Systems, and also the member of IEEE, IEEE Computer Society, IEEE TC on ECBS, and ACM.

Ilija V. Basicevic received his B.Sc.Eng, M.Sc, and PhD degrees from the Faculty of Technical Sciences of the University of Novi Sad in 1998, 2001, and 2009 respectively. His major field of study was computer engineering.

He is currently assistant professor at the Faculty of Technical Sciences. He has published more than 30 papers in journals and conferences. His research interests are in the area of network communication protocols.

Dr. Basicevic is member of IEEE and ACM.

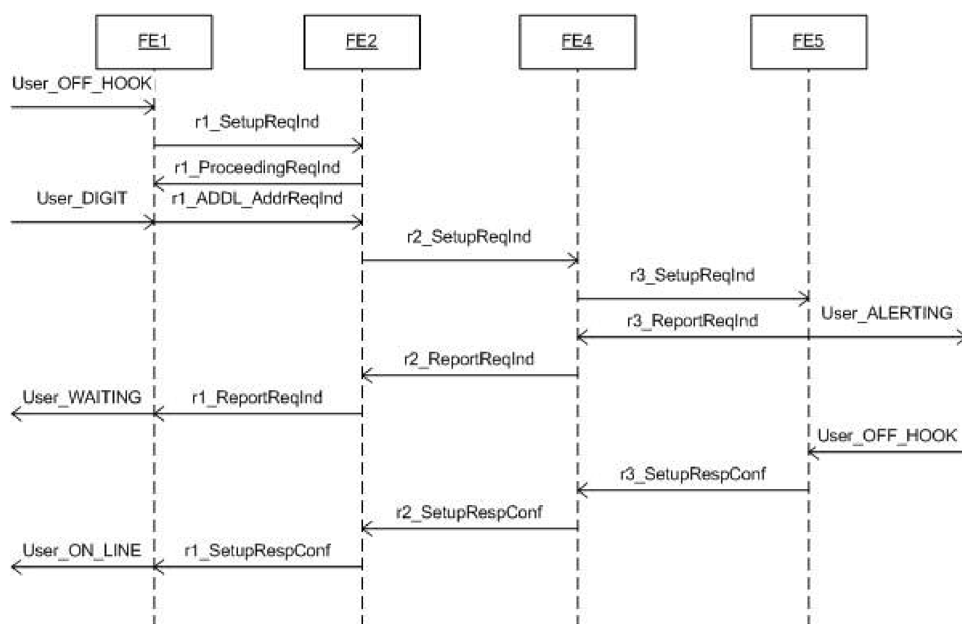

Fig. 2 The message sequence during the successful call establishment

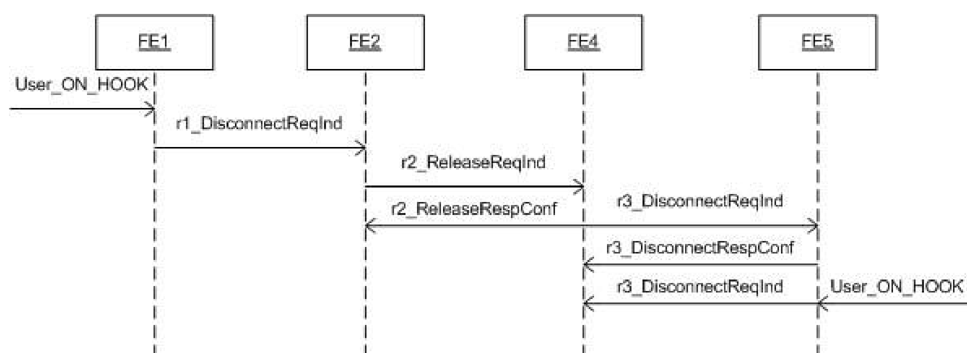

Fig. 3 The message sequence during the successful call release