Verification of a real-time safety-critical protocol using a modelling language with formal data and behaviour semantics

Tamás Tóth, András Vörös, and István Majzik

Budapest University of Technology and Economics, Hungary

Abstract. Formal methods have an important role in ensuring the correctness of safety critical systems. Formal verification techniques could also improve the certification process by providing exact proofs of the expected properties of the design or implementation. However, the application of formal methods in industry is always cumbersome. The lack of experts and the complexity of formal languages prevents the efficient application of formal verification. In this paper we take a step in the direction of using formal methods by introducing a formal modelling framework which helps designers to construct formal models efficiently. In addition, we introduce a toolchain to provide formal verification. Finally, we demonstrate the usefulness of our approach in an industrial case study.

1 Introduction

Nowadays, an ever increasing number of information systems are embedded systems that have a dedicated function in a specific, often safety critical application environment (e.g., components of a railway control system). In case of safety critical systems, failures may endanger human life, or result in serious environmental or material damage, thus ensuring conformance to a correct specification is crucial for their development.

To guarantee that a system operates according to its specification, formal verification techniques can be used. These techniques are based on formal representation of both systems and their properties (requirements), which makes it possible to apply mathematical reasoning to investigate their relationship. Moreover, these methods allow verification of systems in an early phase of the development life cycle.

Since behavior of safety critical systems is often time dependent, the notion of time has to be represented in their models. The most prevalent way to model timed systems is the formalism of timed automata. However, this formalism is only suitable to describe timed behavior with respect to constant values, thus its expressive power is not sufficient to model systems with parametric behavior. Parametric timed automata, an extension of the original formalism, addresses this problem.

Authors' manuscript. Published in Bondavalli A., Ceccarelli A., Ortmeier F. (eds): *Computer Safety, Reliability, and Security*, LNCS 8696, 2014. The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-10557-4_24`.

In this paper we introduce a formal modelling framework for supporting the efficient development of parametric timed automaton based formal models. The modelling language is essentially based on the language of the well-know *Symbolic Analysis Laboratory* (SAL) framework[1] with extensions to simplify the work of the modellers. These extensions enable the modelling of time dependent behaviour on language level.

In the following, first we introduce the main features of language by modelling the development version of an industrial protocol. Then we present our model checking workflow and demonstrate the feasibility of it by the verification of the protocol.

Related Work Our work is inspired by the *SAL* model checker [5] and its language (our extensions are introduced in Section 3). The SAL language enables compact modeling of systems in terms of (unlabeled) symbolic transition systems, however it doesn't support explicit modeling of time related behavior. The aim was to preserve compatibility so that the timed models of our extended language can still be transformed to the input of SAL. As another related tool, *UPPAAL* [1] is a model checker widely used for the verification of timed systems. It has a graphical interface and it provides efficient model checking algorithms to verify timed automata. UPPAAL models can also be transformed to our language with some restrictions: our formalism does not handle complex function declarations. Our approach has different strengths as the underlying Satisfiability Modulo Theories (SMT) technologies are efficient for even complex data structures of the modelled systems. In addition, complex synchronisation constraints can be compactly expressed in our approach. The industrial case study we use was first introduced in [6], where the SAL model checker was used for the verification. Our paper now is based on the lessons learnt from that work.

2 The ProSigma SCAN protocol

ProSigma is a microcontroller based system being developed by Prolan Ltd. Its primary role is to provide reliable communication between the modules of a railway control system. Since this functionality is highly safety critical, it has to be implemented on the highest safety integrity level, SIL4. The system consists of so-called ProSigma devices, that are interconnected in an IP based network. Each ProSigma device contains a so-called ETH unit that is responsible for data transport and so-called LG units (in object modules) that are responsible for handling field objects such as switches and signals. Communication between these units is based on the proprietary SCAN protocol. A part of the messages is forwarded within a ProSigma device, while the other messages are sent through the ETH unit to another ProSigma device addressed by the message.

A ProSigma application typically consists of a control side (e.g., a device at a supervisory control system) and an object side (e.g., a device at a field object).

[1] http://sal.csl.sri.com/

During connection handling, a connection is established and maintained between an LG unit of the control side and an LG unit of the object side. Connection handling includes the following two tasks:

- Establishing the connection: a link is built between the two sides.
- Object state transfer: each side sends its state to the other side.

During connection handling, the state of each connection has to be kept track of at both sides. A connection is defined by the following:

- The connected pairs - the field side has at most four, while the control side has at most one pair.
- The ETH unit providing the connection.

The connection is alive if and only if it is alive through at least one ETH unit. In the first design of the protocol, the connection handling was characterized by the following properties:

The connection is handled via OBJ messages (*OBJ1*, *OBJ2*, *OBJUP*, *OBJDOWN*). Received OBJ messages are processed on two levels. On the first level, based on the timestamps provided in the message, it has to be checked whether the message can be accepted to establish, respectively maintain the connection. On the second level, the acceptability of the object state has to be examined (the object state is more up-to-date than the one accepted last time, it is sent by the right unit, etc.).

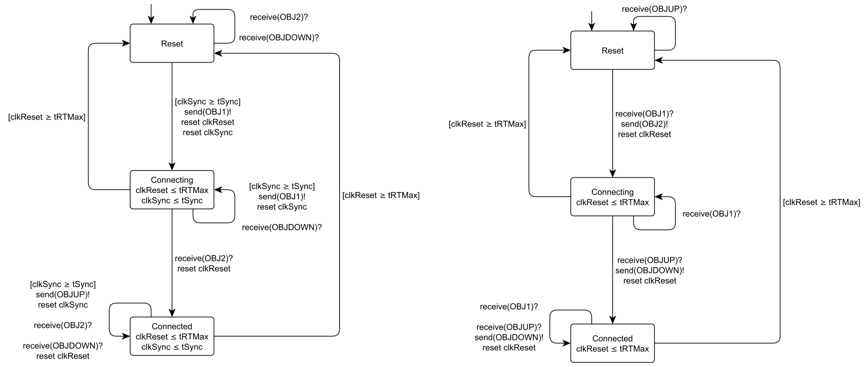
The connection is established in a two-way handshake: the first message is initiating the connection, and the response to it serves as an acknowledgement. In particular, on the field side, the connection is initiated by sending *OBJ1* and is acknowledged by receiving *OBJ2*. On the control side, sending *OBJ2* initiates and receiving *OBJUP* acknowledges the connection.

LG units on the field side must establish the connection with all corresponding control sides through each ETH units independently.

Object state is transferred via two messages (*OBJUP*, *OBJDOWN*). Sending and receiving object state on an ETH channel is possible only after the LG units of the field side and the control side established connection through the ETH channel. The transfer is triggered by the ETH unit of the field side via periodic *TIMESYNC* messages to the corresponding LG units.

The following model (Figure 1.) represents a connection as administered on the field and the control side, respectively, as a network of timed automata with an extended syntax that admits intuitive manipulation of data structures and handshake over multiple input and output labels.

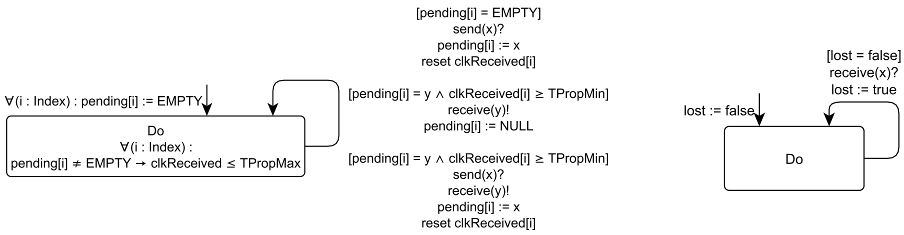
In the model, all messages originate from either of the sides, and propagate for at least *TPropMin* and at most *TPropMax* time units. Assuming that the other parameters of the system are proportional to the propagation time, it will never be the case that a message or the object state stored in it is not acceptable due to an outdated timestamp. Thus in order to simplify the model, the timestamps of messages and acceptability checks are not included in the model (although both the altered timed automata formalism and the specification language enable their modeling).



(a) Field side

(b) Control side

Fig. 1. Connection handling on the field and control side



(a) Channel

(b) Fault model

Fig. 2. Models of the channel and the possible fault occurrences

Channel: The channel serves as a communication medium between components. Its model (Figure 2(a)) has a role to store and delay messages for a certain amount of time specified by its parameters, and then dispatch them.

Initially, the channel is empty. If a message is dispatched and the channel is not full, then the message is stored. If the delay of a message is over, and some component is able to receive it, then the message is forwarded to that component. If the component instantly sends a response, then the response is stored in place of the forwarded message.

Field side: The model of the field side is presented on Figure 1(a).

On the field side, the connection is initially in state *Reset*. As described above, the connection establishment phase is initiated by receiving a *TIMESYNC* message from the ETH unit. For the sake of simplicity, receiving a *TIMESYNC* message is modeled with a clock *clkSync* and corresponding invariants. When the clock reaches *clkSync*, the field module sends *OBJ1* to the control side and traverses to state *Connecting*.

In state *Connecting*, the LG unit waits for *tRtMax* time units for the *OBJ2* message acknowledging the connection. If received in time, the connection is set to state *Connected*, and the object state transfer phase starts on the field side. Otherwise, as clock *clkReset* reaches *tRTMax*, the connection resets.

Object state transfer is also synchronized by *TIMESYNC* messages. When received, the field LG unit sends its state to the control side in a message *OBJUP*. If the time since the last *OBJDOWN* message received reaches *tRtMax*, the connection resets on the field side.

Control side: Similarly to connection handling on the field side, initially, the connection is in state *Reset*. When receiving *OBJ1*, the LG unit sends an *OBJ2* message as response and sets the connection state to *Connecting* (Figure 1(b)) .

State *Connecting* is maintained for at most *tRtMax* time units. If no *OBJUP* is received in that interval, the connection state is set to *Reset*. For an *OBJUP* message received in time, the control LG unit sends its object state in an *OBJDOWN* message, and sets the connection to *Connected*. If the time since the last *OBJUP* message received reaches *tRtMax*, the connection state is set to *Reset*.

Fault model: Since the system has to operate in a safety critical setting, guarantees have to be given for scenarios including unexpected events. For that purpose, the model is extended first with a simple fault model that describes loss of a single message in the channel. The fault model is depicted on Figure 2(b).

3 The Language

The following section demonstrates the capabilities of our extended modelling language by a step-by-step description of the model of the connection handling in the SCAN protocol.

All elements of the model are encapsulated in the context of a *specification*. A specification can have parameters that can be assigned concrete values later on. In this case, a parameter n is introduced to represent the capacity of the channel.

```
specification ProSigmaSpec(n : natural) {
  // Type definitions
  // Constant definitions
  // Function definitions
  // Constraint definitions
  // System definitions
  // Property definitions
}
```

To model automata locations and messages, two enumeration types are defined. Moreover, a type for timing parameters is introduced as the set of non-negative reals, and a subrange is introduced to serve as an index for messages in the channel.

```
type Location : enum { reset, connecting, connected};
type Message : enum { null, obj1, obj2, objup, objdown };
type Param : { x : real | x >= 0 };
type Index : [1 to n];
```

Other supported data types include function, array, tuple and records types, that can be combined to form more complex types.

Timing parameters can then be conveniently modeled as constants of type *Param*. To represent a concrete setting of parameters, a value is assigned to each constant.

```
const tRTMax : Param := 6.0;
const tSync : Param := 3.0;
const tPropMin : Param := 1.0;
const tPropMax : Param := 1.0;
```

However, parameters do not need to be defined explicitly, but can also be represented by constraints expressing their relationships:

```
constraint 2 * tPropMax <= tRTMax;
```

Representing the model of the channel using the language is straightforward. The channel has inputs *msgEvent* and *send* that together model the synchronization primitive $send(x)$. Received messages are stored in array *pending*, until clock *clkReceived* reaches the timeout interval. The synchronization primitive $receive(x)$ is modelled by output *receive*.

```
system Channel := {
  input var msgEvent : boolean;
  input var send : Message;
  global var pending : array Index of Message;
  global var clkReceived : array Index of clock;
  output var receive : Message;
  ...
}
```

The invariant that ensures that messages are delayed for at most *tPropMax* time units can be modelled explicitly.

```
invariant forall (i : Index) : (
  pending[i] = ::null imply clkReceived[i] <= tPropMax
);
```

Analogously to invariant constraints, also urgency constraints can be specified that prohibit time to elapse in certain states.

Initial values of the variables are assigned in an *initialization* section.

```
initialization {
  let receive = ::null;
  let pending = [ i : Index | ::null ];
}
```

Similarly, transitions of the automaton can be mapped to *transition* sections. For example, the transition that dispatches a message and stores the immediate response in a single step can be modelled as follows.

```
transition async (i : Index) :
  msgEvent’ and send’ /= ::null and
  pending[i] /= ::null and clkReceived[i] >= tPropMin --> {
    let pending’[i] = send’;
    let clkReceived’[i] = 0.0;
    let receive’ = pending[i];
  }
```

The keyword *async* denotes that there is an instance of the transition for each index i. The formula followed by `-->` serves as the guard of the guarded command. Apostrophes mark the next state of a variable.

Transitions of the modules can then be defined in a way that complements the definition of the channel. For example, the transition of the Control module that models receiving an *OBJ1* message in location *Reset* can be specified as follows:

```
transition control_location = ::reset and
  receive’ = ::obj1 --> {
    let msgEvent’ = true;
    let send’ = ::obj2;
    let control_location’ = ::connecting;
    let control_reset’= 0.0;
  }
```

Here, *receive* is an input variable, whereas *send* and *msgEvent* are global variables. Notice that the model is modular in the sense that responsibilities of communicating modules and the channel are completely separated. Given all modules are defined, the system can be composed in the following way:

```
system ProSigma :=
  (FieldLG [] ControlLG [] FaultModel) || Channel;
```

That is, communicating modules are composed asynchronously, then the result is composed synchronously to the channel.

Temporal properties can then be defined over the composed system. For example, the model can be validated by checking the property expressing that the capacity of the array for received messages is sufficiently big:

```
property capacity : ProSigma models
  G exists (i : Index) : pending[i] = ::null;
```

The liveness property that expresses that eventually a stable connection is established is the following – note that the usual temporal operators **G** (globally) and **F** (eventually) are used:

```
property live : ProSigma models F G (
  field_location = ::connected and
  control_location = ::connected
)
```

4 The Verification Workflow

The semantics of the language is provided by a series of simplifying model transformations, and a mapping to an SMT problem. Figure 3 depicts the verification workflow. Dashed lines sign the possible extensions with other modelling and verification technologies.

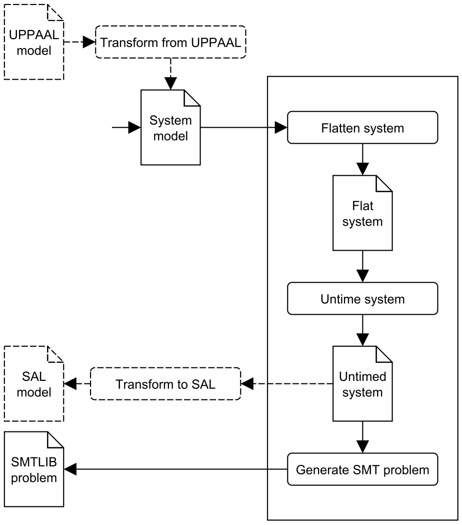


Fig. 3. Verification workflow

The starting point of the workflow is a model in the above language given either directly, or as a result of a transformation from other timed formalisms, e.g. *UPPAAL* [1].

As a first step, the system is automatically flattened, that is, the result of a synchronous, respectively asynchronous composition is established. This is performed by merging the variables, invariant and urgency constraints, and initialization and transition definitions of the components. For example, the following transition of the system *ProSigma* is a result of merging the two transitions presented in the previous section:

```
transition control_location = ::reset and receiveˆ = ::obj1 and
  msgEvent’ and send’ /= ::null and
  pending[1] /= ::null and clkReceived[1] >= tPropMin --> {
    let pending’[1] = send’;
    let clkReceived’[1] = 0.0;
    let send’ = ::obj2;
    let control_location’ = ::connecting;
    let control_reset’ = 0.0;
    let msgEvent’ = true;
    let receive’ = pending[1];
  }
```

During this step, many of the constructed transitions can be eliminated by simply checking the satisfiability of their guards with a call to the underlying SMT solver [4].

In the next step, the model is automatically ”untimed” by expressing the semantics of delay transitions explicitly:

```
transition control_location = ::reset and receiveˆ = ::obj1 and
  msgEvent’ and send’ /=::null and
  pending[1] /= ::null and clkReceived[1] + d >= tPropMin --> {
    let field_location’ = field_location;
    let field_reset’ = field_reset + d;
    let field_sync’ = field_sync + d;
    let send’ = ::obj2;
    let msgEvent’ = true;
    let control_location’ = ::connecting;
    let control_reset’ = 0.0;
    let receive’ = pending[1];
    let pending’ = [ i : Index |
      if i = 1 then send’ else pending[i]
    ];
    let clkReceived’ = [ i : Index |
      if i = 1 then 0.0 else clkReceived[i] + d
    ];
}
```

Here, a combined transition semantics [3][5] is considered, where a transition merges the effects of a delay transition, followed by a discrete transition. For that purpose, a new input variable d is introduced to represent time delay.

Such an untimed system model can easily be mapped to SAL or other intermediate formalisms. At the same time, transition (and initialization) definitions are completed, that is, assignments for unmodified variables (e.g., variables of asynchronous components) are made explicit. As a result, each variable (even those of some complex data type) is assigned a value in at most one assignment of a behavior definition.

The symbolic transition system represented by the model can then be easily expressed in SMT by transforming initialization and transition definitions of the system to predicates $I(\bar{x})$, respectively $T(\bar{x}, \bar{x}')$ as usual [2].

The tooling is implemented in *Eclipse*[2] using *Eclipse Modeling Framework*[3] and relating technologies.

- *Abstract syntax.* The abstract syntax of the language is implemented as a metamodel in *EMF*. It is defined as an extension of the core language suitable for defining complex data types and expressions.
- *Concrete syntax.* The textual concrete syntax is defined by an LL^*-parsable grammar. The textual editor is then generated using the *Xtext*[4] tooling.
- *Semantics.* Model transformations that define the semantics of the language are implemented in *Xtend*[5].
- *Well-formedness rules.* Together with other validation constraints, algorithms for type checking and type inference are implemented for the type system of the language.

5 Verification of the Protocol

To check the properties of the first design, bounded model checking in depth $k = 18$ was executed. As a result, the counterexample depicted on Figure 4. arised. On the diagram, the direction of the arrows symbolizes processing order of events occurring at the same time.

The counterexample reveals that if events occur in a particular order, the loss of even a single message can keep the connection from getting established, contrary to expectations. Due to delay and jitter in IP networks, a situation like that is possible even in practice. The situation can be prevented by modifying the control module so that it responds to *OBJ1* with *OBJ2* in state *Connecting*. The design models were updated accordingly, and the avoidance of this situation was proven by repeated verification.

6 Evaluation and Conclusion

This paper is one step towards scalable formal modelling. We proposed a modelling language to provide better support for the designers of formal models by

[2] `http://eclipse.org/`

[3] `http://eclipse.org/modeling/emf/`

[4] `http://eclipse.org/Xtext/`

[5] `http://eclipse.org/xtend/`

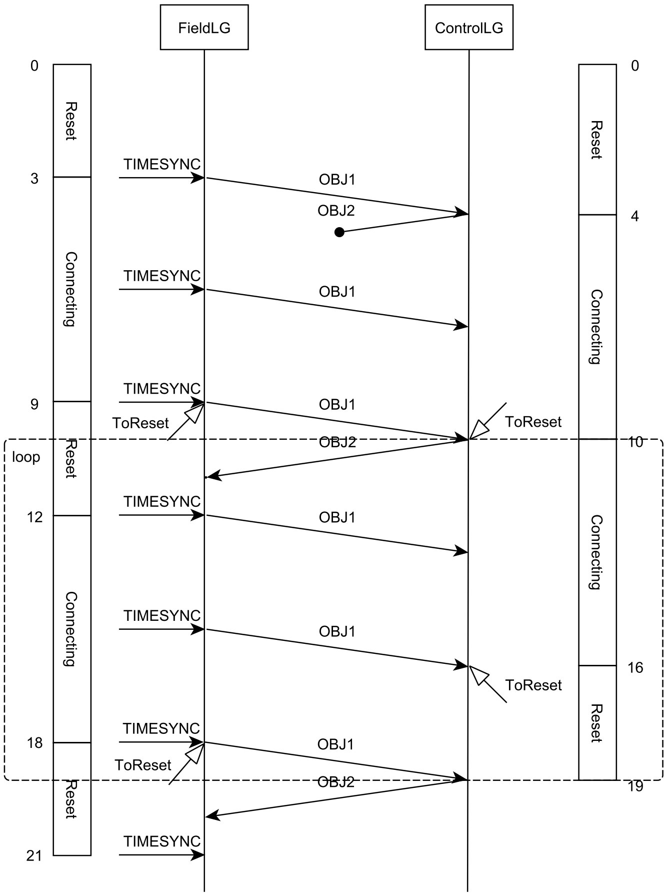


Fig. 4. Counterexample for the liveness property

focusing on the aspects of data semantics, time dependent behaviour, parameterization, and synchronous and asynchronous composition of components. Instead of manually coding flat transition systems, we provided automated model transformations from our extended language to more simple transition systems that can be directly mapped to the input of existing SMT solvers. This way and automated verification workflow is offered.

To evaluate the effectiveness of our language, the formal model of the introduced case study was developed in both our and the SAL languages. Comparing the results, the complexity of the developed models are the following:

The SAL model contains:

- 5 components for modelling the basic behaviour consisting of 410 lines of code,

- 2 components for supporting the proper analysis of the temporal logic specification consisting of 105 lines of code,
- 3 components for recognizing the loops in the state space (required for the verification) consisting of 145 lines of code.

The formal model in our extended language contains 4 components and 235 lines of code, which demonstrates that the new language has its advantage. Moreover, it does not require additional components for the analysis.

Compared to UPPAAL timed automata, the main advantage of our language is the greater flexibility in the handling of clock variables and clock constraints. However, as UPPAAL provides a graphical modelling interface, in case of small models it makes the development of formal models more simple. Regarding the efficiency of verification, both approaches have their strengths.

In the future we plan to further improve our language with higher level modelling constructs. We also plan to develop new model checking algorithms based on induction techniques.

Acknowledgement

This work and the collaboration with *Prolan Ltd.* has been supported by the project *CErtification of CRItical Systems (CECRIS, http://www.cecris-project.eu/), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) nr* 324334, within the context of the *EU Seventh Framework Programme* (FP7).

References

1. Behrmann, G., David, A., Larsen, K.G., Möller, O., Pettersson, P., Yi, W.: Uppaal - present and future. In: Proc. of 40*th* IEEE Conference on Decision and Control. IEEE Computer Society Press (2001)
2. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
3. Kindermann, R., Junttila, T., Niemel, I.: Smt-based induction methods for timed systems. In: Jurdziski, M., Nikovi, D. (eds.) Formal Modeling and Analysis of Timed Systems, LNCS, vol. 7595, pp. 171–187. Springer Berlin Heidelberg (2012)
4. Moura, L., Björner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008)
5. Pike, L.: Real-time system verification by k-induction. Tech. Rep. TM-2005-213751, NASA Langley Research Center (May 2005)
6. Tóth, T., Vörös, A., Majzik, I.: K-induction based verification of real-time safety critical systems. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) New Results in Dependability and Computer Systems, Advances in Intelligent Systems and Computing, vol. 224, pp. 469–478. Springer International Publishing (2013)