Aiding Modular Design and Verification of Safety-Critical Time-Triggered Systems by Use of Executable Formal Specifications*

Kohei Sakurai
Hitachi Europe GmbH, Germany
kohei.sakurai@hase.hitachi-eu.com

Péter Bokor† and Neeraj Suri
Technische Universität Darmstadt, Germany
{pbokor,suri}@cs.tu-darmstadt.de

Abstract

Designing safety-critical systems is a complex process, and especially when the design is carried out at different levels of abstraction where the correctness of the design at one level is not automatically sustained over the next level. In this work we focus on time-triggered (TT) systems where the resources of communication and computation are shared among different applications to reduce the overall cost of the system. This entails serializing both communication and computation which does not necessarily meet the assumptions made by the application. Hence, we present the concept of executable formal specification of general TT systems to establish a faithful model of the TT characteristics. Our focus is on general applications running in a synchronous environment. The proposed model can be easily customized by the user and it is able to support simulation and verification of the system. It also aids the effective deployment of applications, and the validation of the real system with model-based test generation. Our case study shows how the general model can be implemented in the SAL language and how SAL's tool suite can be used to guide the design of general TT systems.

1. Introduction

The design of safety-critical systems entails ensuring predictability of the system's behavior and the assurance of its overall correctness. The *time-triggered* (TT) paradigm has emerged as a viable concept to implement safety-critical systems, with implementations such as TTA [14], FlexRay [24] or SAFEbus [10] actually deployed in the avionic and automotive fields. The use of *formal methods* is increasingly being advocated for the verification of general safety-critical systems (e.g., [20]). However, it has been shown in previous work [4, 21] that for TT systems the correctness of a high-level application does not directly imply the correctness of its implementation. Consequently, formal techniques dedicated to time-triggered systems are needed especially given their increasing deployment. Results about successful formal analysis of TT systems do exist; however, they present specific solutions (e.g., [25, 21, 4]) where modeling patterns can only partially be re-used in new projects. Therefore, we propose a generalized customizable model of TT applications which is not restricted to any dedicated implementation and can be used by practitioners in the field. Our model is an executable specification of the system which (a) not only enables verification but, (b) through simulation of the system, can also guide the deployment of applications, and (c) the effective generation of test suites.

*Research supported in part by Hitachi, EC ReSIST and Genesys
†Also with FTSR Group, Budapest University of Technology and Economics, Hungary

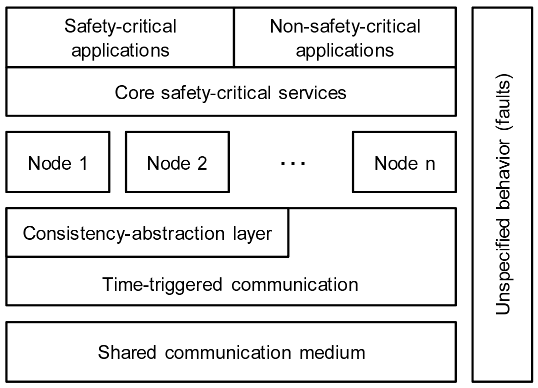
Safety-critical applications
Non-safety-critical applications
Core safety-critical services
Node 1
Node 2
···
Node n
Consistency-abstraction layer
Time-triggered communication
Shared communication medium
Unspecified behavior (faults)

Figure 1. Scheme of TT architectures

To establish the context and contributions of this paper, we provide a brief overview on TT, discuss the motivations and highlight the proposed solution.

The System

Our model (Figure 1) follows the general view of TT systems [22] and augments it by a consistency abstraction layer [23, 4]. User applications and core services are implemented by jobs, each of them running on one or more nodes. The actions of the system are triggered by the passage of time. The execution of jobs in the host nodes is scheduled at design time to guarantee predictability. Nodes communicate with each other using a shared bus such that a communication controller grants write access to the nodes in a round-robin manner and that receiver nodes can read the bus in the time period of message sending. However, this communication pattern can entail inconsistency among replicated jobs running on different nodes. For example, if jobs A and B run respectively before and after job C is scheduled to send a message, job A and B will have a different local views. This mismatch between parallel and sequential sending and processing of messages can be tackled by buffering the messages to delay these operations [23, 4]. In general, it is the designer's decision whether to use this abstraction or not. As the system also hosts safety-critical applications, our view must also incorporate faults at different levels of the architecture.

The Motivation

We seek a unified and formal treatment of general TT systems to guide key system design tasks such as *task scheduling*, *test case generation* and *verification* over user-guided application and system configuration scenarios. Consequently, the user should be able to customize the general model to describe specific applications while ensuring the desired system assurance levels.

A Solution

Deductive reasoning (e.g., theorem proving [20]) is a powerful tool to verify complex properties of even infinite systems; however, it cannot directly be used to simulate the system for finding certain execution paths (e.g., counterexamples, test cases). Therefore, we propose the use of an executable system specification to provide further features besides verification. Since our approach uses the same model to perform different tasks of design and verification, we need not prove conformance between different representations. Our model is easy-to-understand as it maps each component of the high-level model (see Figure 1) into a syntactic module of the applied formal language. To customize the general model, the user only needs to tailor the corresponding modules.

Our overall contributions are:

- We present an *executable* formal model of general TT systems. The model is intuitive and easily *customizable* (for the TT operations and the desired classes of faults) given its *modular* structure. We use the abstraction of discrete time scale which directly stems from the assumption that the system is synchronous.
- We show a *prototype implementation* of the general model by using the SAL language.
- We demonstrate the usability of our prototype by utilizing the tools in the SAL suite to perform, based on the same model, *verification*, effective *deployment* and *test generation* for a case study.

The paper is structured as follows: Section 2 provides an overview of TT systems. Section 3 proposes the executable, modular model of general TT systems and Section 4 shows a prototype implementation of the general model using the SAL language. In Section 5, the example use cases are outlined in the SAL environment. Finally, we discuss related work (Section 6) and conclusions (Section 7).

2. An Overview of TT Systems

This section gives an overview of TT systems [22, 14] and defines a general class of faults.

2.1. Basic Concepts and Definitions

A system consists of N *nodes* with unique IDs $\{1, ..., N\}$. Each node hosts one or more *jobs* that use the local resources of the node when executed. Jobs communicate with each other following a synchronous schedule called TDMA (Time Division Multiple Access). The main idea is that nodes share a communication bus in a round-robin manner[1]. Each node is assigned a time window, called sending *slot*, in each TDMA *round*. Node i sends a message at sending slot i and other nodes can receive this message by identifying the sender by its sending time. The communication is time-triggered because the action of message sending and receipt is launched by the time of local clocks. Collision on the bus is avoided by assuming that clocks are synchronized. Disallowed access to the bus is avoided by so called bus guardians which physically prevent a faulty node from accessing the bus.

Besides the TDMA communication schedule, each node has its own *internal schedule* which determines when jobs are executed. Both the TDMA and the internal node schedules are statically defined at design time. Figure 2 shows an example of both schedules.

2.2. Consistency-Abstraction Layer

Fault-tolerance is often achieved by replicating application jobs on different nodes. A convenient abstraction layer is provided by a mechanism called read/send alignment

[1] Other TT systems, like *frame-based* systems [13], use dedicated channels and can be treated as a special case of our general model.

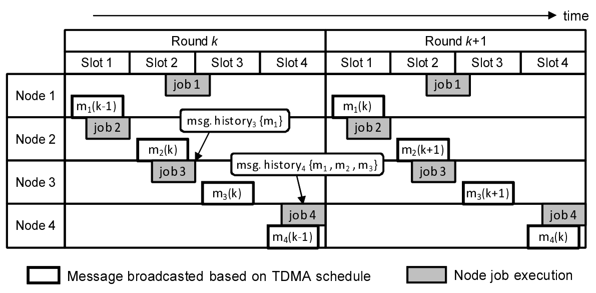

Figure 2. TDMA communication and internal node schedule

[4, 23] which enables nodes to exchange and compute messages as if there were dedicated links between every pair of nodes and the replicated jobs were executed parallel in time. This facilitates the development of applications where replicated jobs are assumed to maintain a common consistent state (e.g., diagnosis [23]). The solution is flexible since it is able to provide the same abstraction independent of how a replicated job is scheduled within the host node. Consequently, even core (transparent) services that should not assume constrained scheduling can be implemented by this technique. For example, a low-level diagnostic service can be designed without posing any assumption about *when* the code of the service is executed within a particular node.

Example Inconsistency We demonstrate the need of abstraction through a simple example of a replicated service. Assume that replication is achieved by jobs that execute the same deterministic operation in every round using the same inputs from other jobs sent via messages. Consistency is defined by requiring that every replica job has the same local state in every round after execution. The main problem in TT systems is that the *freshness* of data sent or processed by different replicas might be different. For example, replicated job 4 reads messages m_1, m_2, and m_3 sent in the current round by nodes 1, 2 and 3 respectively (see Figure 2). On the other hand, job 3 can read only one fresh message, m_1. In this way, inconsistency can arise at round k since jobs 3 and 4 update their local states based on different message histories. Node scheduling also determines when the message calculated by a job can actually be sent. Freshness now means whether a message can be sent in the TDMA round when it is calculated. For example, messages m_1 and m_4 contain the results calculated in the previous round as the job execution of the respective node is after or during the assigned sending slot. On the other hand, node 2 and 3 can send fresh messages m_2 and m_3 in the current round since both jobs complete before the sending slots of the hosting nodes. As a result, messages that are sent in the same TDMA round by different nodes might refer to different TDMA rounds. This can cause inconsistency, if, for example, nodes want to agree on a view of the system regarding the time period of a round (e.g., diagnosis or membership [23]).

General Solution To rectify the previous inconsistencies, we use *read and send alignment* [23, 4] and provide it as a layer between jobs and the host nodes communicating via TDMA. The read alignment layer buffers the messages read from the shared bus and computes a consistent message history in the following way. Assume that a job can read the messages sent by nodes $1, ..., i$ in the current TDMA round. A consistent message history now contains $\{m_1, ..., m_i\}$ as read in the previous TDMA round, and $\{m_{i+1}, ..., m_N\}$ as read in the current round. The send alignment layer buffers the message calculated by the job and sends the old message if there is at least one other job which cannot send the newly computed message. Note that the alignment mechanisms are based on *a priori* known schedules.

2.3. Characterization of Faults

Faults can manifest in any component of the system. The most obvious classification of faults in TT systems distinguishes between communication and application faults. Application faults happen during the execution of the job and are usually specific to the application logic. Faults in communication mean that a message other than intended is sent or received. We define communication faults independent of the application. Note that, in general, the number of faults tolerated by the application is based on their degree of severity [28]. The different classes of communication faults are defined with ascending severity:

- *benign fault*: fault is locally detectable by every receiver other than the sender, e.g., missing message,
- *symmetric value fault*: all receivers read the same semantically incorrect but locally undetectable message,
- *asymmetric value fault* (or Byzantine [16]): the most severe fault where no assumption is made about what message is sent by a faulty sender. We categorize every fault that is neither benign nor symmetric as Byzantine.

Application faults depend on the application itself. The rationale of modeling communication and application faults simultaneously is the ability to analyze their interplay with respect to the high-level specification of the system.

3. A Customizable Formal Model

The aforementioned description cannot directly be used for formal analysis of TT systems. In this section, we propose a template for the formal modeling of general TT

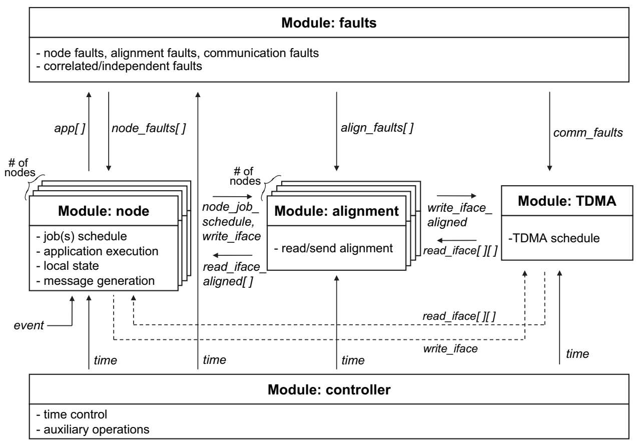

Figure 3. General modular structure of the proposed Customizable Formal Model

systems. The model consists of five types of high-level elements (called modules): *controller*, *node*, *alignment*, *TDMA*, and *faults*. The modules and their interconnections are depicted in Figure 3. Next, we detail the module operations and describe their interfaces. A prototype implementation of the model will be shown in Section 4.

Controller Module The controller module implements the notion of *time* and distributes it to the other modules to orchestrate the synchronized execution of the system. The controller adjusts the time and triggers operations in other modules. Such a centralized treatment of time corresponds to the assumption of synchronized clocks in TT systems.

Node Module The node module has multiple instances, one for each node. This module is in charge of executing the hosted jobs at the time when they are scheduled, updating the local state of each job and handling external events, e.g., reading sensors. For simplicity of further discussion we can assume the every node hosts a single job. The input interface defines events (*event*), the current time and the messages received from other nodes. The messages are accessed by reading a buffer, called read interface, which stores messages read from the bus. If a job uses the consistency abstraction, it reads consistently aligned messages (*read_iface_aligned[]*), otherwise it simply reads data from the TDMA bus (*read_iface[]*). Jobs send messages by writing them into the write interface which are then copied and sent on the bus by the communication controller. The output interface of the node module contains the job schedule (*node_job_schedule*) and the messages sent by a job (*write_iface*). The former one is required by the alignment layer, the latter is sent directly on the bus or via alignment.

Alignment Module The alignment module is a consistency abstraction that implements read and send alignment at every node. The input interface contains the current time, the node's internal schedule, the message to be sent and the messages read on the TDMA bus. Based on this information, the module returns the aligned message of the node (*write_iface_aligned*) and, in the output interface, the consistently aligned incoming messages.

TDMA Module The TDMA module simulates the TDMA communication bus which nodes use to send and receive messages. The input interface consists of the current time and the message to send by each node, and the output interface returns the values of delivered messages at each node (*read_iface[][]*). The returned variable is a matrix of messages where the i^{th} message in the j^{th} row is the message that node i receives from node j. Messages are passed between the node and TDMA modules either directly or via the abstraction layer.

Faults Module The fault module extends the input and output interfaces of normal operation. Modeling of correlated faults needs coordination among different system elements, therefore, we assume that faults are implemented by a single module. The fault module takes the current time as input and injects faults into nodes (*node_faults[]*), the alignment layer (*align_faults[]*) and the communication bus (*comm_faults*). Since the application logic is system specific, the list of jobs (*app[]*) is passed to the fault module to derive application faults.

The proposed overall model supports reusability and customization for varied TT functionalities and applications. The implementation of the modules can be tailored to the characteristics of the actual TT system. For example, when we verify fault-tolerance under different fault conditions within the same TT application, only the fault module needs to be modified. Note that more simplistic models can be proposed by assuming services like alignment or membership. For example, it was shown that the frame-based model can be used to model general TT systems if alignment is used [4]. In this paper, on the contrary, we propose a *realistic* model which can also be used for the design of new algorithms that exploit the characteristics of the TT architecture.

4. An Implementation with the SAL Language

In this section, we make a detailed walkthrough of our prototype implementation of the general formal model described in Section 3. Although we concentrate on safety-critical applications where jobs are replicated and each job execute the same program, our implementation can be easily customized to describe any TT application. We use the SAL language because of its expressiveness and high-level constructs and also because a powerful execution environment is attached to the language [6] which enables direct analysis as shown in Section 5. We describe the implementation of each module and the composition of the modules. The SAL implementation is depicted in a tabular notation. We use the convention that SAL code in the explanation text is written in *italics* type.

Snippet 1: Type declarations, auxiliary functions

```
1  tt{;N: natural, A: natural}: CONTEXT =
2  BEGIN
3    node: TYPE = [1..N];
4    discr_time: TYPE = [0..N];
5    fault: TYPE = {nonfaulty, benign, symmetric, asymmetric};
6    fault_vector: TYPE = ARRAY node OF fault;
7    error: natural = 2;
8    message: TYPE = [0..error];
9    message_array: TYPE = ARRAY node OF message;
10   function:TYPE =[[message_array,message,message]–> message];
11
12   %auxiliary definitions
13   _fault_counter(v: fault_vector,e: fault,sum: [0..N],i: node):[0..N]=
14     IF i = 0 THEN sum ELSE
15     _fault_counter(v,e,sum+IF v[i]=e THEN 1 ELSE 0 ENDIF,i–1)
16     ENDIF;
```

SAL is a typed language and every SAL model begins with the definitions of types and functions (Snippet 1). Our general system model contains N nodes (*node* at line 3), the usual communication faults (*fault* at line 5) and a binary message domain which is augmented with an error value (*message* at line 8). Time is modeled on discrete scale such that each clock tick corresponds to a slot (*discr_time* at line 4). The idea is that the same clock tick triggers, in every module, all operations that are supposed to be performed in the corresponding slot, i.e., scheduled jobs are executed (in node), messages are written/read to/from the bus (in TDMA), messages are buffered (in alignment) and faults are generated (in faults). A virtual slot is defined (with time value 0) to model jobs that can read every message from the previous TDMA round and are ready to send in the current one from slot 1 on. The correctness of our discrete-time abstraction stems directly from the assumption that the precision of the applied clock synchronization algorithm allows agreement on the time slots. Otherwise, the discrete-time model needs to be justified by the user (e.g., [25]). In our model, jobs execute applications (*function* at line 10) which are functions taking a message received from each other node, the current local state and an external event as inputs and returning the new value of the local state. For simplicity, messages, states and events share the same type. In addition, array types define a value for each node (e.g., *message_array* at line 9). These basic definitions can be customized by the user. For example, a simple counter function is defined (lines 13-16) which returns the number of faults in an array.

Snippet 2: Controller module

```
17 controller: MODULE =
18 BEGIN
19   INPUT
20     inp_ev_vec: ARRAY node OF message
21   OUTPUT
22     time: discr_time,
23     fun: function,
24     ev_vec: ARRAY node OF message
25   INITIALIZATION
26     time = 0;
27   DEFINITION
28     ev_vec = inp_ev_vec;
29   TRANSITION
30     [
31       time < N --> time' = time + 1;
32     []
33       time = N --> time' = 0;
34     []
35       ELSE -->
36     ]
37 END;
```

SAL is able to directly map the modules of the general model into modules of the language. SAL modules define local variables, communicate with other modules via input and output variables, initialize local and output variables, define invariants and guarded transitions. The controller module (Snippet 2) adjusts the discrete time in a periodic manner, thus, modeling an infinite sequence of TDMA rounds. In the SAL implementation, the module also maintains auxiliary operations like the definition of the replicated application logic and the distribution of node events. Assuming that a job is a replicated instance of a safety-critical application, the application logic (*fun* at line 23) can be defined only once and passed on to each node. Since *fun* is not initialized, SAL will arbitrarily assign a function to it. Events are defined as external input variables (*inp_ev_vec* at line 20) which are passed to the nodes (*ev_vec* at lines 24,28) for processing.

SAL allows parameterized definition of modules. The parameters need to be defined when the module definitions are used to compose the system. An alignment module instance (Snippet 3) is defined for each node (*id* at line 38). We abstract the job schedule of a node by defining at which discrete time instant the job starts executing (*job_sched* at line 45) and whether the nodes of the system are able to send the latest message in the current TDMA round (*send_curr_round_vec* at line 48). Local buffers are defined to store values that were sent by other nodes (*read_iface_buffered* at line 41) and that were computed by the corresponding node (*write_iface_buffered* at line 42) in the previous TDMA round. The former one is

Snippet 3: Alignment module

```
38 alignment[id: node]: MODULE =
39 BEGIN
40   LOCAL
41     read_iface_buffered: message_array,
42     write_iface_buffered: message
43   INPUT
44     time: discr_time,
45     job_sched: discr_time,
46     read_iface: message_array,
47     write_iface: message,
48     send_curr_round_vec: ARRAY node OF BOOLEAN
49   OUTPUT
50     read_iface_aligned: message_array,
51     write_iface_aligned: message
52   DEFINITION
53     write_iface_aligned =     %send alignment
54       IF FORALL (n: node): send_curr_round_vec[n]
55         THEN write_iface ELSE
56           IF time > job_sched  %job exec. is modeled as atomic event
57             THEN write_iface_buffered ELSE write_iface
58           ENDIF
59       ENDIF;
60     read_iface_aligned = [[n:node]    %read alignment
61       IF n < job_sched
62         THEN read_iface_buffered[n] ELSE read_iface[n]
63       ENDIF]
64   INITIALIZATION
65     read_iface_buffered = [[n:node] 0];
66     write_iface_buffered = 0;
67   TRANSITION
68     [
69       time > 0 -->
70         read_iface_buffered'[time] = read_iface[time];
71         write_iface_buffered' =
72           IF time = job_sched
73             THEN write_iface ELSE write_iface_buffered
74           ENDIF;
75     []
76       ELSE -->
77     ]
78 END;
```

Snippet 4: Node module

```
79  node[id: node]: MODULE =
80  BEGIN
81    INPUT
82      time: discr_time,
83      fun: function,
84      read_iface_aligned: message_array,
85      ev: message
86    OUTPUT
87      write_iface: message,
88      job_sched: discr_time,
89      send_curr_round: BOOLEAN,
90      local_state: message
91    INITIALIZATION
92      local_state = 0;
93      send_curr_round IN {v: BOOLEAN |
94        IF job_sched >= id THEN v = FALSE ELSE TRUE ENDIF};
95    DEFINITION
96      write_iface = local_state; %assumption: node sends local state
97    TRANSITION
98      [
99        time = job_sched -->
100         local_state' = fun(read_iface_aligned, local_state, ev);
101     []
102       ELSE -->
103     ]
104 END;
```

updated every time a remote node sends a message on the bus (line 70), the latter one only changes when the local node updates its local state and generates a new message (lines 71-74). Send (lines 53-59) and read alignment (lines 60-63) can be defined as invariants based on the current and buffered values. Definitions in SAL (labeled by DEFINITION) can be thought of as macros that use the values of other state variables. Consequently, using SAL definitions is not only a logical way of modeling send and read alignment but, as definitions do not affect the state transition relation of the system, it can also save state space during the analysis.

The node module (Snippet 4) is also parameterized by the identifier of the node (*id* at line 79). The job schedule is not initialized which corresponds to an arbitrary schedule. This is in accordance with the premise that core (safety-critical) services are not prioritized. Unrealistic cases are ruled out such that a node is only able to send the fresh message in the same TDMA round if it finishes execution before its sending slot (lines 93-94)[2]. We abstract that the application is executed instantaneously at the time when the job is scheduled (99-100). For simplicity, it is assumed that the local state is sent as the node's message (line 96). However, in general, the message can be a function of the local state. Note that the domain of discrete time instants $[0..N]$ cannot cover the full generality of job scheduling. It cannot be modeled that a node reads everything up to time i (including the message sent in slot i) and sends immediately at time $(i+1)$. For that, the model needs to be extended such that an intermediate time instant is defined between i and $(i+1)$ where the application computes the message to send. Such extension can affect the complexity of the analysis; therefore, its use is only recommended if needed.

The TDMA module is responsible for modeling the communication of messages on the bus (Snippet 5). In general, faults are defined and propagated by the fault module. However, our SAL implementation reduces the number of transitions by directly injecting communication faults in the TDMA module. The output of the module is a matrix which indicates the value received by node i from node j after each tick. If node j is not the sender in the slot then the value is unchanged (line 120). Otherwise, the non-faulty and faulty cases are distinguished. If the sender is non-faulty, the correct value that is determined by the alignment layer is sent (lines 121-122). In case of a benign sender, every recipient receives *error* (lines 123-124)[3], while symmetric senders

[2]We use SAL's `IN` operator to non-deterministically assign a value from a set of constrained candidates; the empty constraint is denoted by the Boolean `TRUE`.

[3]The definition can be modified such that the faulty sender can read its

Snippet 5: TDMA module

```
105 TDMA: MODULE =
106 BEGIN
107   INPUT
108     time: discr_time,
109     write_iface_aligned_vec: message_array,
110     fv_comm: fault_vector
111   OUTPUT
112     read_iface_vec: ARRAY node OF message_array
113   INITIALIZATION
114     read_iface_vec = [[n:node] [[m:node] 0]];
115   TRANSITION
116     [
117       time > 0 -->
118         read_iface_vec' IN {v: ARRAY node OF message_array |
119           FORALL (i, j: node):
120             IF j /= time THEN v[i][j]=read_iface_vec[i][j] ELSE (
121             IF fv_comm[j] = nonfaulty
122              THEN v[i][j] = write_iface_aligned_vec[j] ELSE (
123             IF fv_comm[j] = benign
124              THEN v[i][j] = error ELSE (
125             IF fv_comm[j] = symmetric
126              THEN FORALL(k: node):
127                 v[i][j] = v[k][j] AND v[i][j] /= error ELSE
128             TRUE
129             ENDIF) ENDIF) ENDIF) ENDIF};
130     []
131       ELSE -->
132     ]
133 END;
```

distribute arbitrary but consistent and valid data (lines 125-127). Asymmetric senders can send any value to any node (line 128).

Snippet 6: Comm_faults module

```
134 comm_faults: MODULE =
135 BEGIN
136   INPUT
137     time: discr_time
138   OUTPUT
139     fv_comm: fault_vector
140   INITIALIZATION
141     %at most A asymmetric faults
142     fv_comm IN {v: fault_vector |
143         _fault_counter(v, asymmetric, 0, N) <= A};
144   TRANSITION
145     [
146       time = 0 -->
147         fv_comm' IN {v: fault_vector |
148             _fault_counter(v, asymmetric, 0, N) <= A};
149     []
150       ELSE -->
151     ]
152 END;
```

Our prototype currently implements communication faults (Snippet 6) and other desired system specific faults can be added by the user. Usually, the number of faults that the application is able to tolerate is limited. We show, on the example of asymmetric value faults, how the number of communication faults can be tuned in our model. Initially, the number of asymmetric faults is limited by A (lines 142-143) which is an input parameter of the model (line 1). Transient faults can be modeled by the periodic re-definition of the fault vector (lines 147-148).

The previous modules can be composed together easily by wiring the corresponding input and output variables (see Snippet 7 in the Appendix). We use the synchronous composition operator ($||$) since we want that modules execute transitions in parallel. This means, for example, that the simulated execution of the application occurs parallel with sending a message on the bus. Recall that, although "hard-wired" in this prototype implementation, the use of the alignment module is, in general, optional.

5. Example Use Cases of Executable Formal Specification: Design and Verification

We show how the SAL model of TT systems can be used to verify properties about the system (Section 5.1), to find appropriate scheduling of jobs (Section 5.2) and to automatically generate test cases for specific test goals (Section 5.3). The different tools we use are all part of the SAL environment; thus, they can directly work on the model described in the previous section. In every case, the execution engine is a *model checker* which performs exhaustive simulation of the system model [5]. Independent of the actual model checking algorithm, we only assume that the model checker is able to explore all executions of the system. Therefore, we can safely state that a property is true in a system if the model checker cannot show a counterexample. For simplicity, we assume that the system contains four nodes ($N = 4$). Note that setting N to a constant value is necessary in classical model checking as it is impossible to explore infinite many states.

5.1. Verification

Task: Consistent Replica States Suppose we want to prove that send and read alignment indeed implements the abstraction of dedicated communication paths and parallel job execution. We consider an abstract application which is implemented by an *arbitrary* function taking N messages and the local state as inputs and returning the new local state. Similarly to the type of *fun* (see Snippet 1), all values are ternary (0, 1 and *error*). The correctness of the abstraction can be shown by proving consistency, i.e., replica jobs have the same state at the end of each round[4]:

```
consistency: THEOREM
 system |- G(time=0 => FORALL(i,j:node):
  local_state_vec[i]=local_state_vec[j]);
```

own message.

[4]The auxiliary variables `local_state_vec[i]` (and `local_state_prev_vec[i]`) were introduced to denote the local state of job i at the current (and previous) round.

Result: Consistency in Symmetric Systems In fact, the SAL model checker could prove the property unless asymmetric faults are allowed in the system. Since *fun* was never explicitly initialized in the model, SAL assumes that it can be any function and tries all possibilities. This corresponds to checking that `consistency` is true for any application expressed by *fun*. Note that this is a special case of the general theorem which states that consistency holds for any application with any number of replica nodes [4]. The model checker finds a counterexample if an asymmetric sender distributes different messages to different nodes which then compute inconsistent local states. This means that `consistency` can be proven for $A = 0$ and a counterexample is found if $A > 0$. In the second case, consistency can only be obtained through the use of a Byzantine agreement protocol [16].

Discussion: Complexity We observed that it is computationally very expensive to consider all possible evaluation of *fun*. However, we could successfully prove the property within a few tens of minutes with the BMC (Bounded Model Checker) of SAL. The proof took approximately 20 minutes running on a single processor of a double-core Intel Xeon 5130 at 2 GHz with 4 GBytes memory. SAL was installed on a Linux system with kernel version 2.6.17. We remark that the BDD (Binary Decision Diagrams) model checker turned out to be ineffective to prove the property as computing the BDD-representation of the model took a long time ($>$ 1 hour).

The next applications of the model are based on finding counterexamples which are in general computationally less complex than complete verification. This is because only a portion of the state space must be explored. In fact, proving consistency took the most time in our set of experiments.

5.2. Scheduling of Jobs

Task: Schedule for Reduced Abstraction Delay We now show a proof-of-concept example of how to use the model checker to find effective scheduling of replicated jobs within the hosting nodes. We saw that the delay induced by the alignment abstraction can be mitigated if all nodes are able to send a message in the same round when the message is computed. In this case the delay is caused by the read alignment and is one TDMA round. In fact, the model checker can find a proper schedule which minimizes these delays. We achieve this by stating that such a schedule does not exist and the model checker finds a counterexample which is a correct solution. To track the delay between sending a message and processing it at a remote node, we extend the domain of messages and define that *fun* returns a special value `SPECV` if all input messages are 1. The following property states that it is never true that the local state in a round is `SPECV` and the local state in the previous round is 1 in all nodes.

```
exist_schedule:
 THEOREM system |-
  G(NOT(time=0 AND FORALL(i:node):
   local_state_prev_vec[i]=1 =>
   EXISTS(j:node):local_state_vec[j]=SPECV));
```

Result: Early Scheduled Jobs The property cannot be proven and a counterexample is provided where the four jobs are executed "early", at slots 0, 0, 1 and 1 respectively. We can see that this schedule indeed allows every job to send the fresh message (value 1), therefore, the overall delay can be reduced in this proof-of-concept example. Note that the same technique can be used to find a proper mapping to deploy jobs on nodes even if more complex constraints are specified.

5.3. Test Generation

The idea of automated test generation is to construct a sequence of inputs (called *test case*) that will cause the system under test to exhibit some behavior of interest, called *test goal*. Model checkers can be naturally used to generate test cases such that Boolean *trap variables* are defined which are initially false and set to true by the program when the corresponding test goal is reached. The model checker is instructed to prove that the trap variables cannot become true; therefore, every counterexample is a valid test case. However, the straightforward way of doing that can be ineffective. For example, the strategy of generating one test case for each test goal might be redundant. More sophisticated techniques leverage the model checker, for example, by checking for paths which are extensions (i.e., continuations) of already explored executions [9]. This technique is also implemented by SAL's ATG (Automated Test Generation) tool which wraps the BMC and BDD model checkers of SAL by augmenting them with clever techniques needed for effective test generation.

Task: Tests for Specific Message Patterns We used the ATG tool of SAL to generate a sequence of events observed by a node which drives the system to a certain state. We re-define *fun* that it returns `SPECV` if a node observes the external event 1. Let the test goal be reaching a state where the value `SPECV` is sent on the bus as the message of the 1^{st} node. Therefore, we place the following assignment of trap variable `atg_trap` (newly added) in the TDMA module:

```
atg_trap'=IF EXISTS(n:node):
 read_iface_vec[n][1]=SPECV THEN TRUE
 ELSE FALSE ENDIF;
```

Result: Test Case Found By running SAL ATG, we get that node 1 observes event 1 and generates `SPECV` but it cannot send it immediately because of the node schedule. Therefore, the trap variable only becomes true in the second TDMA round of the execution path which constitutes the test case. In the SAL example, a sequence of events, containing one event for every node, is returned by the tool for each slot until the trap variable becomes true.

We have shown how our executable formal specification and trap variables can be used to support model-based test generation. An important issue about testing is the coverage of the obtained test suite, i.e., whether it is able to exercise the system to the desired extent determined by the coverage metrics. The discussion about test coverage in TT systems is beyond the scope of this paper.

6. Related Work

Formal methods have been successfully used for the verification of various TT applications. For example, the membership protocol in the TTP/C time-triggered protocol suite was shown to be correct by hand proofs in [3], or automated theorem proving was applied to analyze an agreement protocol in another time-triggered environment [21]. Other work used executable formal specification to model check the startup protocol in TTP/C [25]. It is also possible to specify the system in an intermediate, preferably understandable and easy-to-read, notation and translate it into the input language of the verification engine (e.g., [17]). Our prototype implementation omits this intermediate step and specifies the system directly in the input language of the analysis. We argue that the modularity of the proposed model of TT systems, and the resemblance of its structure with the real system fulfills the role of a precise though intuitive specification. Our approach mainly differs from previous work in that it proposes a skeleton model of the target systems which can be used as a template for customized solutions and the specification needs not be created from scratch.

As part of the system integration process, deployment, allocation and scheduling tasks can be uniformly thought of as restrictions with respect to the unconstrained space of solutions. Different techniques such as constraint propagation, branch and bound, backtracking or mixed integer programming have been proposed (e.g., [15, 12, 2]), however, they require either the development of new computation engines (e.g., written in C [12] or Java [15]) or the use of existing dedicated engines (e.g., [2]) to solve (or even optimize) the constraint problem. Although our method, when used for scheduling, might be outperformed by other techniques, it uses the same system representation that is used for other tasks. Therefore, the overall cost of design and analysis can be reduced. If the constraint problem entails infeasible complexity with our executable formal model, the application of other techniques is inevitable. However, it is possible that the combination of our approach with other techniques enhances the quality and performance of finding the best solution. The investigation of using model checking for system integration is part of our ongoing work.

Testing of distributed systems deals with *what* to test and *how* to test. Our approach is related to the first one because we generate test cases as opposed to actually testing the system. In distributed system, the second question is nontrivial due to issues such as interoperability, synchronization, timing, and concurrency. Model-based testing can feed existing solutions for testing distributed systems like [1, 27].

We apply the method of model-based test generation using a model checker [7, 19] to generate tests for the validation of TT systems. The idea is to challenge the model checker to find execution paths in the model of the system that reach specific test goals. This is done by stating that no such path exists and the counterexample returned by the model checker can be directly used as a test case. As an alternative or supplementary to the model-based approach, requirement-based test-case generation [29, 18] can be used where tests are generated by analyzing the requirements only. This can be useful if tests are required to be independent of the unit under test.

7. Conclusion

We have proposed a formal but intuitive method to support the development of safety-critical TT systems from design to verification and testing. The solution is general and the core elements of the method (i.e., system characteristics, application logic and fault modes) can be customized by the application developer. In our case study, we have implemented a prototype model of general safety-critical TT applications. This model has been analyzed by model checkers and simple verification, task scheduling and test generation examples have been shown. We used the SAL language in our implementation, but other executable specification languages such as NuSMV [26], SystemC [11] can be equally applicable. However, additional transformation steps might be needed to translate from the specification to the language of the execution engine (e.g., model checking of SystemC codes [8]). As compared to previous work, the main strength of our method is that it provides an all-in-one solution which can be used for tasks that usually require multiple representations of the system.

In future work, we plan to analyze real time-triggered protocols with our method. We also look at deriving coverage metrics for testing TT applications and we look for new solutions of hardware/software integration in safety-critical TT systems. In particular, we are interested in techniques that have been proven to be applicable for standard model-checking (e.g., SAT solvers) but have not yet been used for system integration.

References

[1] G. A. Alvarez and F. Cristian. Simulation-Based Testing of Communication Protocols for Dependable Embedded Systems. *J. Supercomput.*, 16(1-2):93–116, 2000.

[2] A. Balogh, A. Pataricza, and J. Rácz. Scheduling of Embedded Time-Triggered Systems. In *Proc. Workshop on Engineering Fault Tolerant Systems*, pp. 44–49, 2007.

[3] G. Bauer et al. An Investigation of Membership and Clique Avoidance in TTP/C. In *Proc. SRDS*, pp. 118–124, 2000.

[4] P. Bokor et al. Sustaining Property Verification of Synchronous Dependable Protocols over Implementation. In *Proc. HASE*, pp. 169–178, 2007.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[6] L. de Moura et al. SAL 2. In *Proc. Computer Aided Verification*, pp. 496–500, 2004.

[7] A. Gargantini and C. L. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In Proc. European Software Eng. Conf., pp. 146–162, 1999.

[8] A. Habibi and S. Tahar. An Approach for the Verification of SystemC Designs using AsmL. In *Proc. Automated Technology for Verification and Analysis*, pp. 69–83, 2005.

[9] G. Hamon, L. de Moura, and J. Rushby. Generating Efficient Test Sets with a Model Checker. In *Proc. SW Eng. and Formal Methods*, pp. 261–270, 2004.

[10] K. Hoyme and K. Driscoll. SAFEbus. *Aerospace and Electronic Systems Magazine, IEEE*, 8(3):34–39, 1993.

[11] Open SystemC Initiative. http://www.systemc.org.

[12] S. Islam and N. Suri. A Multi Variable Optimization Approach for the Design of Integrated Dependable Real-Time Embedded Systems. In *Proc. Embedded and Ubiquitous Computing*, pp. 517–530, 2007.

[13] R. M. Kieckhafer et al. The MAFT Architecture for Distributed Fault Tolerance. *IEEE Trans. Comput.*, 37(4):398–405, 1988.

[14] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, Jan. 2003.

[15] K. Kuchcinski. Constraints-Driven Scheduling and Resource Assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, 2003.

[16] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Prog. Lang. and Sys.*, 4(3), 1982.

[17] S. P. Miller et all. Proving the Shalls. In *Proc. Formal Methods Europe*, pp. 75–93, 2003.

[18] A. Rajan, M. W. Whalen, and M. P. Heimdahl. Model Validation using Automatically Generated Requirements-Based Tests. In *Proc. HASE*, pp. 95–104, 2007.

[19] S. Rayadurgam and M. Heimdahl. Coverage based Test-Case Generation using Model Checkers. In *Proc. Workshop on Eng. of Comp. Based Systems*, pp. 83–91, 2001.

[20] J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. TR SRI-CSL-95-1, SRI Int., 1995.

[21] J. Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Trans. Softw. Eng.*, 25(5):651–660, 1999.

[22] J. Rushby. Bus Architectures for Safety-Critical Embedded Systems. In *Proc. Embedded Software*, pp. 306–323, 2001.

[23] M. Serafini et al. A Tunable Add-On Diagnostic Protocol for Time Triggered Systems. In *Proc. DSN*, pp. 164–174, 2007.

[24] FlexRay Specification. http://www.flexray.com.

[25] W. Steiner et al. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *Proc. DSN*, pp. 189–198, 2004.

[26] NuSMV Toolset. http://nusmv.irst.itc.it/.

[27] W. Tsai et al. Scenario-based Object-Oriented Test Frameworks for Testing Distributed Systems. In *Proc. Future Trends of Distr. Computing Sys.*, pp. 288–294, 2003.

[28] C. J. Walter, M. Hugue, and N. Suri. Continual On-Line Diagnosis of Hybrid Faults. In *Proc. 4th Conf. on Dep. Computing for Critical Applications*, pp. 233–249, 1994.

[29] M. W. Whalen, A. Rajan, M. P. Heimdahl, and S. P. Miller. Coverage Metrics for Requirements-based Testing. In *Proc. Software Testing and Analysis*, pp. 25–36, 2006.

Appendix

Snippet 7: Synchr. composition of N-node system

```
system: MODULE =
  controller
  | | (WITH INPUT read_iface_aligned_vec:
                              ARRAY node OF message_array
      WITH INPUT ev_vec: ARRAY node OF message
      WITH OUTPUT write_iface_vec: message_array
      WITH OUTPUT job_sched_vec: ARRAY node OF discr_time
      WITH OUTPUT send_curr_round_vec:
                              ARRAY node OF BOOLEAN
      WITH OUTPUT local_state_vec: ARRAY node OF message
      (| | (n: node): RENAME
           read_iface_aligned TO read_iface_aligned_vec[n],
           ev TO ev_vec[n],
           write_iface TO write_iface_vec[n],
           job_sched TO job_sched_vec[n],
           send_curr_round TO send_curr_round_vec[n],
           local_state TO local_state_vec[n]
      IN node[n]))
  | | (WITH INPUT job_sched_vec: ARRAY node OF discr_time
      WITH INPUT read_iface_vec:
                              ARRAY node OF message_array
      WITH INPUT write_iface_vec: message_array
      WITH OUTPUT read_iface_aligned_vec:
                              ARRAY node OF message_array
      WITH OUTPUT write_iface_aligned_vec: message_array
      (| | (n: node): RENAME
           job_sched TO job_sched_vec[n],
           read_iface TO read_iface_vec[n],
           write_iface TO write_iface_vec[n],
           read_iface_aligned TO read_iface_aligned_vec[n],
           write_iface_aligned TO write_iface_aligned_vec[n]
      IN alignment[n]))
  | | comm_faults
  | | TDMA;
```