Modeling and Verification for Timing Satisfaction of Fault-Tolerant Systems with Finiteness

Chih-Hong Cheng*, Christian Buckl‡, Javier Esparza† Alois Knoll*

*Unit 6: Robotics and Embedded Systems, Department of Informatics, TU Munich, Germany

*Unit 11: Theoretical Computer Science, Department of Informatics, TU Munich, Germany

†Fortiss GmbH, Germany

Email:{chengch,buckl,esparza,knoll}@in.tum.de

Abstract

The increasing use of model-based tools enables further use of formal verification techniques in the context of distributed real-time systems. To avoid state explosion, it is necessary to construct verification models that focus on the aspects under consideration.

In this paper, we discuss how we construct a verification model for timing analysis in distributed real-time systems. We (1) give observations concerning restrictions of timed automata to model these systems, (2) formulate mathematical representations on how to perform model-to-model transformation to derive verification models from system models, and (3) propose some theoretical criteria how to reduce the model size. The latter is in particular important, as for the verification of complex systems, an efficient model reflecting the properties of the system under consideration is equally important to the verification algorithm itself. Finally, we present an extension of the model-based development tool FTOS, designed to develop fault-tolerant systems, to demonstrate our approach.

I. Introduction

The complexity of distributed real-time systems is growing rapidly; model-based development tools are used to accelerate the development process and increase the quality of the produced code. In addition, it is possible to integrate formal verification as analysis technique into these tools.

Currently, the standard verification process is achieved by first translating system models into verification models, followed by verifying relevant properties by verification engines using special algorithms. In the verification community, researchers focus on tighter theoretical complexity bounds or computationally faster algorithms to reduce the required time for verification. Nevertheless, if it comes to verification of complex systems, an efficient model reflecting the properties of the system under consideration becomes essential. By *efficient model*, we refer to a model containing "just-enough" information of the system behavior regarding these properties. In fact, an inefficient modeling with irrelevant details can simply render the verification intractable.

Within this paper, we introduce an approach for the construction of such an efficient model for the verification of timing assumptions and constraints. The approach is presented in, but not restricted to, the context of FTOS [5], a model-based development tool for the design of fault-tolerant systems.

In our presentation, we first introduce FTOS, mention insights regarding differences in comparison to other development tools, and propose our two-phase verification methodology (sec. II). Then based on FTOS and timed automata [3], we describe the model construction process, focusing on the aspects concerning expressiveness, modification, and efficiency.

- **(Expressiveness)** We give observations regarding restrictions of timed automata to construct models of real-time systems (sec. III-A, III-B, III-D); these observations are valid not only in the context of FTOS, but apply also for other systems.
- **(Modification)** We formulate mathematical representations how to perform model modification to derive verification models from system models (sec. III-C).
- **(Efficiency)** With the understanding of (1) complexities of verification and (2) our problem structure, we propose some theoretical criteria regarding how to construct an efficient model, such that it is possible for existing model checkers to generate results within reasonable time (sec. IV).

At last, we report our preliminary implementation (sec. V), mention related work (sec. VI), and conclude this paper (sec. VII).

II. FTOS and Motivating Examples

A. Introduction to FTOS

FTOS is a model-based development tool for the development of fault-tolerant real-time systems, that alleviates designers' burden by offering code generation for non-functional aspects with high extensibility.

The conceptual modeling in FTOS uses multi-aspect techniques comprising four different perspectives:

- **Hardware Model**: The hardware model specifies the hardware used, including specifications of *electronic control units* (ECUs) and the interconnecting *network*.
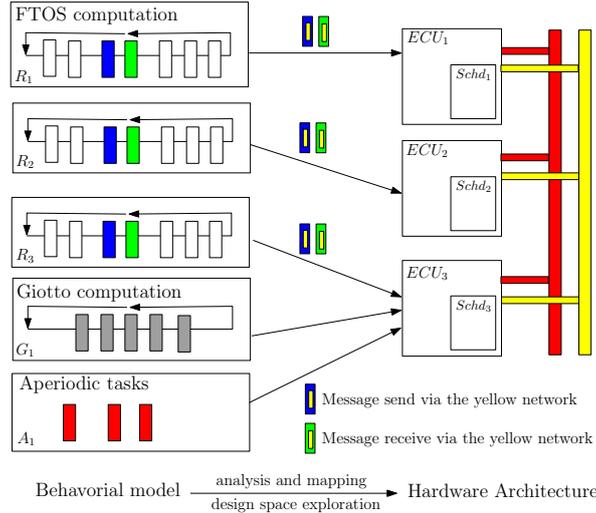
Figure 1.   Behaviorial models and architectures.

- **Software Model**: The underlying model of computation in FTOS shares large similarities with that of Giotto [8], which is based on the concept of Logical Execution Times. A designer should specify *tasks*, *ports*, *inputs*, *outputs*, and *jobs*.
- **Fault Model**: The fault model specifies the *fault hypothesis* of the system, which includes the set of *fault containment units* (FCUs) (possible faults concerning locations, types, durations), and the set of *fault configurations* (possible simultaneously activating FCUs). Examples for the fault hypothesis are:
    1) A network link can have message lost (fault type: MsgLoss) with minimum interval between consecutive occurrences equal to 3 milliseconds[1].
    2) A software task can produce errors (fault type: WrongResult) due to a fault within an associated sensor; once happened, it will not be corrected unless explicitly done by the user or the fault-tolerant mechanism. The minimum interval for the correct operation between two consecutive faults of the sensor is expected to be 500 milliseconds[2].
- **Fault-tolerance Model**: The fault-tolerance model specifies methods to detect errors and to repair and restore the system.

During code generation, FTOS selects, adapts and combines pre-implemented code templates based on model features. A detailed description of FTOS can be found in [5].

B. General Settings and Examples

The concepts presented in this paper do not only apply for FTOS, but also a range of other related projects, such as Giotto [8] or event-driven tasks with fixed deadlines. Figure 1 shows the different models of execution. An aperiodic or sporadic function is event driven; when such an event happens, a deadline is assigned to the task handling the event. Giotto functions are functions that interact synchronously at macro step level (logical level), while at micro step level the execution is asynchronous. For detailed description of Giotto and the concept of logical execution time, see [8]. FTOS functions are extensions of Giotto functions. Intuitively they are equipped with fault-tolerance abilities such that the system can resist faults defined by the fault model. In fig. 1, three redundant copies (R_1, R_2, R_3) are deployed on the three machines (ECU_1, ECU_2, ECU_3).

The figure also shows the necessity of a mapping the behavioral model (in FTOS: software model) to the architecture model (in FTOS: hardware model). Note that in general a design space exploration is needed for finding such a mapping. For details, we refer readers to articles regarding platform-based design [10]. Since this mapping is specified in FTOS by the developer, our analysis can start from a given selection of hardware and software settings.

C. Verification Goals

The main property of fault-tolerant systems that needs to be verified is the ability to withstand the assumed faults. The fault assumptions are summarized in the fault hypothesis (in FTOS: fault model) that defines faults regarding its location, effect, and frequency.

---

[1]This minimum interval is called *least time between faults* (LTBF) in FTOS and is derived from the required probability of the system to withstand the fault.

[2]In (1) the message loss is transient, and in (2) computation errors caused by hardware faults are permanent.
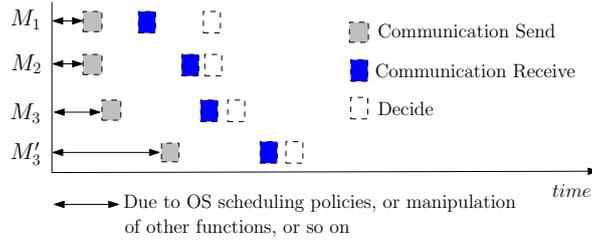
Figure 2. Internal nondeterminism due to scheduling differences.

The verification of such systems is hindered by two aspects: deadline violations and non-determinism due to e.g. imperfect synchronization of redundant units.

1) In ordinary systems, correctness relies on the assumption that a scheduling never leads to deadline violations (without loss of generality, we assume that deadlines specified in our model are hard). Nevertheless, in fault-tolerant systems, the constraint can be loosened. Due to replication, a deadline violation of one unit might be tolerated. In fact, the violation of the deadline can be categorized as an occurrence of a fault defined in the fault model. This brings dramatic differences between fault-tolerant systems and ordinary systems, i.e., deadline violation is feasible or acceptable provided that there exists a fault-tolerance mechanism such that the effect of fault can be eliminated.

2) On the other hand, replication also introduces further difficulties. In ordinary Giotto systems, internal determinism is guaranteed, meaning that two deployments having the same relative ordering in the micro step level will have the same behavior, irrelevent of the absolute timing. Unfortunately, internal determinism will not be maintained if no constraints are added additionally on FTOS functions. Consider fig. 2, where M_1, M_2 and M_3 are three deployments. The send action will broadcast messages to other machines regarding its liveness. Ideally, when no error happens, then each machine should conceive a consistent view of the system. However, when the scheduling of M_3 changes to that of M_3', with zero time transmission, the result will be an inconsistent view at M_1 and M_2. This brings semantic incompatibility between different deployments.

To solve these problems, we thus propose the concept called *deterministic assumption* [6]. Intuitively, the goal is to assume that the implementation of fault tolerance mechanisms will always provide a consistent view for all correct machines regardless of deadline violation and scheduling issues. In practice, this will place constraints regarding the earliest and latest arrival time between messages sent, which need to be verified.

For above purpose, we adapt a two-phase verification process in our tool FTOS-Verify:

- **(Phase 1: Verification on the platform independent layer)** We first assume that the deterministic assumption holds in all deployments. Based on this assumption, we construct a verification model. The model is an abstract machine (closed model) where injection of faults is regulated based on the fault model. The model offers precision by revealing detailed mechanisms of fault-tolerance. Our theoretical foundation enables us to construct a concise model with huge benefits[3]. For this phase, the mathematical formulation and the proof of theorems are stated in [6]; it will not be the focus of this paper.

- **(Phase 2: Validity checking of the behavior-architectural mappings)** In this phase, we have to focus on two aspects. First, we have to check whether the deterministic assumption holds in the platform. Second, we have to check if there exists possibilities where deadlines are violated, and the violation exceeds the constraint specified and regulated in the fault model. Note that since the correctness of the data and mechanisms are checked in the first phase, in the latter phase only protocol checking (timing) is needed. This will be the focus and the main contribution of the paper. For the analysis of the temporal behavior, we transform the models in FTOS to communicating timed automata (CTA). In the following sections, we will describe our observations, relevant parts of the construction process, and theoretical criteria for model efficiency. By using a generalized view, the results are applicable not only in the context of FTOS, but can be used for verifying temporal behavior for generic distributed real-time systems.

III. System Modeling and Observations

We use an extended format of communicating timed automata (CTA) [4] using variables of finite domain to express the features of the behavioral model. It is important to mention that this extended format does not change the expressiveness of CTA.

Definition 1. *A system of communicating timed automata is a tuple* $\mathcal{S} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, *where* $\mathcal{A}_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i, Inv_i)$ *is an automaton with the following constraints.*

---

[3]Our theorem states that we can construct a synchronous verification model (exponentially smaller reachable state space) provided that (1) the deterministic assumption holds and (2) the properties are local (in-machine) LTL properties without using temporal operator **X**. This makes formal verification of large systems practicable.
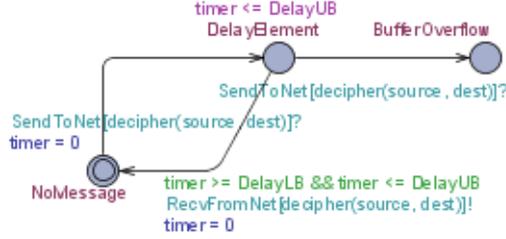
Figure 3.   Timed automaton representing point-to-point transmission with capacity 1.

- Q_i *is a finite set of* modes *(locations).*
- V_i *is the set of finite-domain integer variables.*
- $C_i = \{c_{i_1}, \ldots, c_{i_m}\}$ *is the set of* clock variables.
- $Sync_i = \{s_{i_1}, \ldots, s_{i_n}\}$ *is the set of* synchronizers; *each synchronizer s is of the format $s \in \{?,!\} \times \Sigma$ where elements in Σ represent synchronizer symbols. Conceptually, "?" represents receiving, and "!" represents sending.*
- $q_i \in Q_i$ *is the* initial location *of the automaton.*
- $Jump_i = Q_i \times Guards_i \times Sync_i \to Q_i \times Resets_i$ *is the jump from mode to mode.*
  1) *$Guards_i$ is the conjunction of inequalities of the form $c_{i_x} \sim k$ or $v_{j_x} \sim k'$, where $c_{i_x} \in C_i$, $v_{j_x} \in V_j, j = 1 \ldots n$, $k, k' \in \mathbb{N}$, and $\sim \in \{=, >, <\}$.*
  2) *$Resets_i$ is the set of assignments of the form $c_{i_x} := 0$ or $v_{i_x} := k'$, where $c_{i_x} \in C_i$, $v_{i_x} \in V_i$, and $k' \in \mathbb{N}$.*
- Inv_i *is the set of mode invariants mapping a mode to a subspace of $\mathbb{R}^{|C_i|}$ indicating the possible clock values to maintain in the mode.*

In the following, we summarize required components of the verification model and outline our observations.

A. Network Element with Finite Capacity

To model the network of the distributed system, an appropriate level of detail must be selected. In general, for a network with message delay and n junction points, we have to model such a network with $n(n-1)$ automata to handle point-to-point communication. Fig. 3 is the template (defined in UPPAAL [4]) of a timed automaton which models the point-to-point transmission with storage capacity equal to 1, and one overflow location. The function `decipher(source,dest)` is used to return the index of the channel.

Observation 1. *For modeling of network components, only finite capacity can be reached. Furthermore, the number of controlled locations grows exponentially as the number of allowed storage increases, because the variable delay (due to the fault model) or the routing scheme may lead to an arbitrary ordering of arrived messages[4].*

B. Task Element with Finite Precision

For Giotto-like MoCs, tasks are units which perform dedicated computations. Modeling the task execution can vary based depending on whether the applied scheduling is preemptive. Fig. 4 shows a timed automaton representing the task execution with potential context switches. Since context switch and preemption can occur, the task should keep the record for the remaining time (portion) to finish the task. The variable `percentage` represents the progress of execution and `increment` reflects the minimal advance related to the time accuracy used during verification. The constraints regarding time accuracy imply finite precision in the model.

Observation 2. *Modeling of tasks can only be achieved with finite precision, since with context switch, we need to record the portion of executed tasks. This also brings issues between expressiveness and complexity; a better accuracy regarding the timing behavior of the context switch (with finer time unit) leads to increasing complexity of the resulting model since it depends on the biggest integer used in the system.*

C. Job Processing Element

The task of the job processing element is to manage the execution of tasks and to implement the inter-task communication. The construction in timed automata may vary due to the concrete application. However, for FTOS, a fixed sequence of atomic actions is defined in the software model. Each atomic action can be represented by a similar model as used for the task model described previously. The main difference is that the models of the atomic actions are linked together instead of having a closed loop in the automaton representing the job processing element. Here we omit the

---

[4]As the synchronizer in CTA takes no time, the timing and the ordering of messages should be modeled in the network automaton.
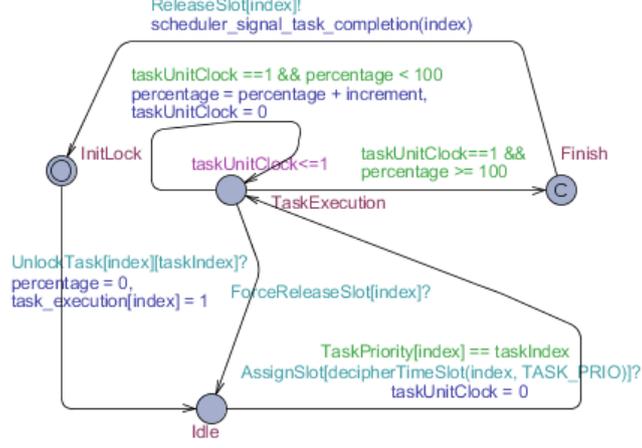
Figure 4. A timed automaton to represent task execution with context switch.

detailed construction process for the original model, but focus on the transformation into the according verification model. We give two motivating examples.

- In order to model the effect of faults, we need to add additional edges on the original model to represent the occurrence of faults.
- To observe deadline violation, additional clocks that reflect the time progress since event occurrence, locations that represent the deadlines, and jumps are required to annotate the original model.

For these purposes, we define this annotation as a sequence of edit-operations over a labeled graph [14]; this facilitates the mathematical formulation how we transform between models.

Definition 2. *Define five atomic edit actions as follows*[5].

1) **Clock add**: *Given a clock variable c, $\lambda X.\text{clock\_add}(X, c)$ is an operation that adds a clock to X. Formally speaking, given $\mathcal{A}_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i, Inv_i)$, the result of $\text{clock\_add}(\mathcal{A}_i, c)$ is a new timed automaton $\mathcal{A}'_i = (Q_i, V_i, C_i \cup \{c\}, Sync_i, q_i, Jump_i, Inv_i)$.*

2) **Variable add**: *Given a variable v, $\lambda X.\text{var\_add}(X, v)$ is an operation that adds a variable to X. Formally speaking, given $\mathcal{A}_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i, Inv_i)$, the result of $\text{var\_add}(\mathcal{A}_i, v)$ is a new timed automaton $\mathcal{A}'_i = (Q_i, V_i \cup \{v\}, C_i, Sync_i, q_i, Jump_i, Inv_i)$.*

3) **Location add** *Given a location q and an invariant inv, where inv is the conjunction of inequalities of the form $c_{i_x} \sim k$ with clock c_{i_x}, $k \in \mathbb{N}$, and $\sim \in \{=, >, <\}$, $\lambda X.\text{vertex\_add}(\mathcal{A}_i, q, inv)$ is an operation that adds a location to X with invariant condition inv. Formally speaking, let $\mathcal{A}_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i, Inv_i)$, the result of $\text{vertex\_add}(\mathcal{A}_i, q, inv)$ is a new timed automaton $\mathcal{A}'_i = (Q_i \cup \{q\}, V_i, C_i, Sync_i, q_i, Jump_i, Inv_i \cup \{inv\})$.*

4) **Jump add**: *Given two locations $q, q' \in Q$ with guard g, assignment a, and set of synchronizers s, where*
   a) *g is the conjunction of inequalities of the form $c_{i_x} \sim k$ or $v_{j_x} \sim k'$, where c_{i_x} is a clock, v_{j_x} is a variable, $k, k' \in \mathbb{N}$, and $\sim \in \{=, >, <\}$.*
   b) *a is the set of assignments of the form $c_{i_x} := 0$ or $v_{i_x} := k'$, where c_{i_x} is a clock, v_{i_x} is a variable, and $k' \in \mathbb{N}$.*

   Let $\mathcal{A}_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i, Inv_i)$, then the result of $\text{jump\_add}(\mathcal{A}_i, q, g, a, s, q')$ is a new timed automaton $\mathcal{A}'_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i \cup \{((q, g, s), (q', a))\}, Inv_i)$ by adding an arc $((q, g, s), (q', a))$ to $Jump_i$.

5) **Jump edit**: *Given two locations $q, q' \in Q$ with guards g, g', assignments a, a', and sets of synchronizers s, s'.*
   a) *g, g' are conjunctions of inequalities of the form $c_{i_x} \sim k$ or $v_{j_x} \sim k'$, where c_{i_x} is a clock, v_{j_x} is a variable, $k, k' \in \mathbb{N}$, and $\sim \in \{=, >, <\}$.*
   b) *a, a' are sets of assignments of the form $c_{i_x} := 0$ or $v_{i_x} := k'$, where c_{i_x} is a clock, v_{i_x} is a variable, and $k' \in \mathbb{N}$.*

   Let $\mathcal{A}_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i, Inv_i)$, then the result of $\text{jump\_edit}(\mathcal{A}_i, q, g, a, s, q', g', a', s')$ is a new timed automaton $\mathcal{A}'_i = (Q_i, V_i, C_i, Sync_i, q_i, Jump_i \cup \{((q, g', s'), (q', a'))\} \setminus \{((q, g, s), (q', a))\}, Inv_i)$ by changing the arc $((q, g, s), (q', a))$ to $((q, g', s'), (q', a'))$ in $Jump_i$.

Note that in our formulations, we assume due to simplification reasons that the added element is not identical to any elements in the original set, and every newly added location or jump is well defined (e.g., to add $\mathcal{A}$ a new location with

---

[5]Here we merely define edit actions necessary for our propositions and algorithms; more can be defined.

invariants using clock c, c should have been defined in $\mathcal{A}$).

Definition 3. *Let an edit sequence be $\hat{e} = e_1 \circ e_2 \ldots \circ e_n$, where $e_1, e_2 \ldots, e_n$ are edit actions. Define the result of $\hat{e}$ on $\mathcal{A}$, in symbols $\mathcal{A}e_1 \circ e_2 \ldots \circ e_n$ inductively as follows.*

- *$\mathcal{A}\epsilon = \mathcal{A}$ where ϵ is the null sequence.*
- *$\forall c,\ \mathcal{A}\left(\lambda X.\text{clock\_add}(X, c)\right) \circ e_2 \ldots \circ e_n = \text{clock\_add}(\mathcal{A}, c) \circ e_2 \ldots \circ e_n.$*
- *$\forall v,\ \mathcal{A}\left(\lambda X.\text{var\_add}(X, v)\right) \circ e_2 \ldots \circ e_n = \text{var\_add}(\mathcal{A}, v) \circ e_2 \ldots \circ e_n.$*
- *$\forall q, L,\ \mathcal{A}\left(\lambda X.\text{vertex\_add}(X, q, L)\right) \circ e_2 \ldots \circ e_n = \text{vertex\_add}(\mathcal{A}, q, L) \circ e_2 \ldots \circ e_n.$*
- *$\forall q, q', a, s, g,\ \mathcal{A}\left(\lambda X.\text{jump\_add}(X, q, g, a, s, q')\right) \circ e_2 \ldots \circ e_n = \text{jump\_add}(\mathcal{A}, q, g, a, s, q') \circ e_2 \ldots \circ e_n.$*
- *$\forall q, q', a, g, s, a', g', s',\ \mathcal{A}(\lambda X.\text{jump\_edit}(X, q, g, a, s, q', g', a', s')) \circ e_2 \ldots \circ e_n = \text{jump\_edit}(\mathcal{A}, q, g, a, s, q', g', a', s') \circ e_2 \ldots \circ e_n.$*

Starting from the textual description of the fault model, we can construct the set of *deadline requirements* $\bigcup_i (q_i, q_i', T_i)$ for the system model $\mathcal{S}$. Intuitively this means that for all runs entering the location q_i, it must subsequently enter q_i' within at most T_i time units. Based on above definitions, we sketch the algorithm[6] how to generate the verification model from the system model as follows:

Algorithm: `GenVerificationModelPart()`
```
{
    /* Input: Original system model S = {A₁,...,Aₙ} */
    /* Output: Verification model Sᵥ */
    let ê = ε.
    forall deadline requirements (qᵢ, qᵢ′, Tᵢ), qᵢ ∈ Qᵢ,
    /* add new clock and new variable for testing */
        ê := ê ∘ λX.clock_add(X, cᵢ).
        ê := ê ∘ λX.var_add(X, vᵢ).
    /* q_{dl.vioᵢ} is the location for deadline violation */
        ê := ê ∘ λX.vertex_add(X, q_{dl.vioᵢ}, φ).
        forall incoming jumps ((q, g, s), (qᵢ, a)) of qᵢ,
            ê := ê ∘ λX.jump_edit(X, q, g, s, qᵢ, a, g, a′, s),
                where a′ = a ∧ (cᵢ := 0) ∧ (vᵢ := 0).
        endfor
        forall incoming jumps ((q, g, s), (qᵢ′, a)) of qᵢ′,
            ê := ê ∘ λX.jump_edit(X, q, g, s, qᵢ′, a, g, a′, s),
                where a′ = a ∪ {(vᵢ := 1)}.
        endfor
        forall reachable locations q from qᵢ,
            ê := ê ∘ λX.jump_add(X, q, g, φ, φ, q_{dl.vioᵢ}).
                where g is defined as (vᵢ = 0) ∧ (cᵢ > Tᵢ).
        endfor
    endfor
    return Sᵥ := Sê. /* apply changes in ê */
}
```

For the property of deterministic assumption mentioned in section II-C, similar algorithms can be applied to annotate clocks, locations, and jumps; the problem for checking deterministic assumption in FTOS turns to be a reachability problem in timed automata.

D. Dispatcher

With respect to the operating system, we have to model the dispatcher explicitly. The modeled dispatcher merely captures the scheme for the execution of threads; deadline violation, fault-tolerance or error handling is modeled in the job processing element. Therefore, it can be used in arbitrary settings and not only in FTOS. Due to different scheduling algorithms, the model of the dispatcher differs dramatically regarding actual verifiability. For our analysis, we use priority based dispatchers modeling either FIFO or round-robin techniques. Nevertheless, as context switch of tasks/threads occurs, we have the following observation.

Observation 3. *Using a round-robin dispatcher leads to exponential increase of possible behaviors compared to a FIFO-based dispatcher with the number of parallel tasks, if no assumptions on the task behavior can be made.*

In summary, this section gave insight in the main components of the verification model and their construction. Besides the job processing element, all components and related observations can be directly applied for arbitrary real-time systems. For the job processing element, we described a generic way to use annotations to construct a model to use for verifying

---

[6]This editing algorithm is not general; for requirement (q_i, q_i', T_i), q_i is not reentered before entering q_i' because actions in the job processing element are chained.
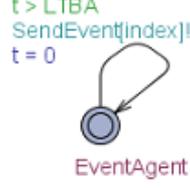
Figure 5.   A sample timed automaton representing the event agent.

the absence of deadline violations. In the next section, we point out how aperiodic behavior introduced by faults or events can be considered.

IV. INVOCATION OF FAULTS AND APERIODIC EVENTS

To perform verification, modeling the arrival of faults or aperiodic events is necessary to establish a closed model, and in this section we consider its effect. In FTOS, the probability of faults is implicitly reflected by the concept called *least time between faults* (LTBF). In our analysis, the invocation of aperiodic tasks can be done similarly - the least time between occurrences of events for aperiodic tasks is defined as *least time between arrivals* (LTBA). With LTBA or LTBF, we can augment the original model with a timed automaton producing the event (called *event agent*) similar to fig. 5. However, since LTBF (or LTBA) is an integer which might be relatively large, and the complexity of verification in timed systems is related to this integer[7], the use of LTBF (or LTBA) may hinder the practicability of model checking. Thus we propose some methods to effectively reduce the value of LTBF (or LTBA) with equivalence criterion. For simplicity reasons, the following theorems are all discussed using event-triggered aperiodic functions with LTBA without loss of generality.

Proposition 1. *Let system S have one FTOS function with periodic deadline T and one event-triggered aperiodic function.*
  - *W.L.O.G., let $A_{event} = (\{q\}, \emptyset, t, \{!event\}, q, \{((q, (t > LTBA_S), \{!event\}), (q, (t := 0))), \phi\})$ be the timed automaton of the event agent, where $LTBA_S = T_S$ be the least time between two consecutive aperiodic events.*

Let T_p be the maximal time interval for the system to finish processing the event (called deadline interval from now on)[8]. If $T_S > T_p + T$, then consider another system S', where $S' =$ jump_edit$(S, q, (t > T_S), q, (t := 0), \{!event\}, (t > T_p + T), (t := 0), \{!event\})$, i.e., the only difference is to change $LTBA_S$ from T_S to $T_p + T$. Then both systems are equivalent regarding their behavior concerning deadline violation. That is, for S and S', either they both satisfy the deadline, or they both miss the deadline.

An intuitive argument for the bound $T_p + T$ can be derived using fig. 6. Tasks and events influence the execution of each other. The execution of an arriving event is influenced by the currently running task. During the deadline interval of the event, this and all preceding tasks are influenced as well. The chain of influence can only be stopped if the execution is decoupled. Since preceding tasks are decoupled by definition, two events with a minimal bound of $T_p + T$ can not influence the execution of the same task. In fig. 6, we call a time point $\hat{t}$ *decoupling point* if two consecutive tasks immediately before and after $\hat{t}$ are not mutually influenced due to the occurrence of an event.

Proof: We consider four possible cases in S':
1) Consider the case where in S', it is proven that no deadline is violated. When the verification engine proofs that the deadline is never violated with $LTBA_{S'} = T_{S'}$ in S', the deadline of the FTOS function in S will never be violated because $T_S > T_{S'}$; the verification engine has already considered all cases in S.
2) Consider the case where in S', the counter-example indicates that the i-th aperiodic task violates the deadline. We further split the discussion in subjects whether it is the first time for S' to process the event. Our goal is to construct a counter-example for deadline violation in S from the counter-example in S'.
   a) If $i = 1$, i.e., it is the first time for S' to execute the aperiodic task, then this deadline violation can also occur in S, since no constraints are made for the first occurrence of events in S or S'.
   b) If $i \neq 1$, consider the $(i - 1)$-th aperiodic execution which does not violate the deadline. Let the time for the coming of event $(i - 1)$-th be t, and let the interval between the $(i - 1)$-th and the i-th event be T'. The system should finish the $(i - 1)$-th processing before time $t + T_p$. Since $T' > LTBA_{S'} = T_p + T$, from time $t + T_p$ to $t + T'$, FTOS function should finish one of its execution and proceed a new one. Let the time for the

---

[7]The reachability problem for timed automata is PSPACE-complete, i.e., the complexity is exponential to (1) the number of clocks and (2) the maximum integer used in the system. Concerning (2), if the maximum number changes from 10 to 100, intuitively the execution time can increase by the factor of k^{90}, where $k > 1$.

[8]Let $t_{arrival}$ be the time for the event arrival. If the system can not finish processing this event within time $t_{arrival} + T_p$, then the system violates the deadline.
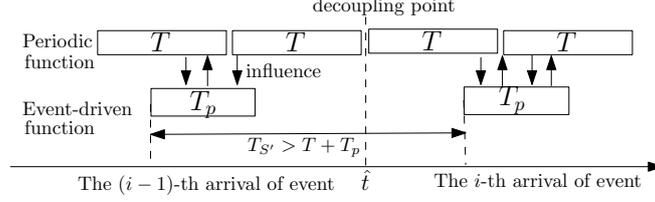
Figure 6. Illustrations for proofs of Proposition 1.

start of that cycle be $\hat{t}$. If we change the counter-example time trace such that no event has happened before $\hat{t}$, we still get a counter-example trace in S'. This new counter-example trace is also a counter-example trace in S.

3) Consider the case where in S', the counter-example indicates that the FTOS function violates the deadline. Let the time which violates the deadline be t (note that t is the multiple of T). Let the occurrence of the nearest event be t' (if there exists no such event, then both S and S' can deadlock).

   a) If $t - t' \geq T_p + T$, then the event is processed before time $t - T$, the starting of the period which violates the deadline. In this way, the system violates the deadline with only the existence of FTOS function, thus in S, the deadline will also be violated.

   b) If $t - t' < T_p + T$, we consider whether the event is the first one being processed.

      i) If yes, then the counter-example in S' is also a counter-example in S.

      ii) If not, then consider the time where the previous event occurs, and let the time be t''. Since $t'' - t' > T + T_p$, we can find a decoupling point $\hat{t}$, where $t'' + T_p \leq \hat{t} \leq t'$, where at $\hat{t}$ it starts a new period. In this way, we can perform the same technique stated in (2-b) before $\hat{t}$.

4) Consider the case where in S', the counter-example indicates that both the i-th aperiodic task and the FTOS function violate the deadline. Let the time which violates the deadline be t (note that t is the multiple of T), then the event occurs in time $t - T_p$. By an argumentation similar to point 3-b, a counter-example trace in S can be established.

∎

Remark: (1) Proposition 1 formulates the insight that previous events occurred long before can not influence the current processing and scheduling, and therefore, are not the root cause of deadline violation. In other words, we could also construct a counter example with a single event as root cause. (2) The introduction of faults can be viewed analogously. For faults, in FTOS (or similar fault-tolerant systems) the value of $T_p + T$ is much smaller then T_S (LTBF), and this brings significant advantages for construction of a model with smaller state space.

Proposition 1 can only be used for very simple systems with only one task and one event. In the following, we will generalize the result to systems consisting of one periodic function with period T and several aperiodic functions.

Proposition 2. *Let S be a system with n aperiodic functions. Each function with index i, where $i = 1 \ldots n$, is associated with a pair $(LTBA_i, T_{p_i}) \in \mathbb{N} \times \mathbb{N}$ describing the LTBA and deadline interval. Consider another system S', where the only difference is to perform the following change: if for all $i = 1 \ldots n$, $LTBA_i > T(\sum_{i=1 \ldots n} \lceil \frac{T_{p_i}}{T} \rceil) + T$, then we change $LTBA_i$ to $T(\sum_{i=1 \ldots n} \lceil \frac{T_{p_i}}{T} \rceil) + T$. Both systems S and S' are equivalent regarding their behavior concerning deadline violation.*

Proof: We consider the following cases.

1) If S' does not violate the deadline, then so does S.

2) Consider the case where in S', the counter-example indicates that the i-th aperiodic task of type j violates the deadline.

   a) If $i \neq 1$, let the time for the i-th and $(i-1)$-th arrival of type-j events be t and t'. Our goal is to find the decoupling point $\hat{t}$ such that we can overlook all previously happened events.
   Since $t - t' > T(\sum_{i=1 \ldots n} \lceil \frac{T_{p_i}}{T} \rceil) + T$, then within $[t', t]$ the periodic function is executed at least $\alpha = (\sum_{i=1 \ldots n} \lceil \frac{T_{p_i}}{T} \rceil + 1) - 1$ times. Consider the worst case where it is only executed α times. Within $[t', t]$, there are $\alpha + 1$ potential decoupling points $t_0, \ldots, t_{\alpha+1}$, where $\forall k = 1 \ldots \alpha + 1$, $t_k - t_{k-1} = T$.
   Due to the sparsity of events, each type of event arrives at most once within $[t', t]$. For each type m, the according event with deadline interval T_{p_m} will overlap in worst case at most $\lceil \frac{T_{p_i}}{T} \rceil$ of these potential decoupling points. Thus the total number of overlapped points is at most $\sum_{i=1 \ldots n} \lceil \frac{T_{p_i}}{T} \rceil = \alpha$, which is less than the number of points among $\{t_0, \ldots, t_{\alpha+1}\}$. Therefore, there exists at least one point $t_g \in \{t_0, \ldots, t_{\alpha+1}\}$ such that it is not overlapped by any deadline interval. Thus we can set the decoupling point $\hat{t}$ as t_g. As

a result, we can construct an equivalent counter-example where no event has happened before $\hat{t}$. This new counter-example trace is also a counter-example trace in S.

b) If $i = 1$, let the time for the i-th arrival of type-j events be t.

   i) If for all type of events, the according events occurred at most once before t, then the counter-example is also a counter-example in S.

   ii) If there exists some type of events occurred more than once: let c_m for type m be the total number of events occurred in the counter-example and t_{c_m} be the latest event arrival time. Choose m' such that $c_m > 1$ and $\forall m = 1 \ldots n, m \neq j, t_{c_{m'}} > t_{c_m}$. Then we can find the decoupling point between the $(c_{m'} - 1)$-th and $(c_{m'})$-th arrival of event with type m', similar to the argument in case 2-a.

3) Consider cases where the deadlock happens in the FTOS function.

a) If in the counter-example no event has occurred, then both S and S' can deadlock.

b) Otherwise, first we try to pick an event based on arguments in 2-b-ii. If possible, then the decoupling point can be found, and the counter-example for S can be established. If selection based on 2-b-ii is not possible, it follows the statement of 2-b-i that the counter example for S' is also a counter-example in S. ∎

Lastly, we discuss the most general case.

Proposition 3. *Let system S have m periodic FTOS/Giotto functions with periodic deadline $T_{f_1}, T_{f_2}, \ldots T_{f_m}$, where $i = 1 \ldots m$, and n aperiodic functions. Each function with index j, where $j = 1 \ldots n$, is associated with a pair $(LTBA_j, T_{p_j}) \in \mathbb{N} \times \mathbb{N}$ describing the LTBA and deadline interval. Consider another system S', where the only difference is to perform the following change: if for all $i = 1 \ldots n$, $LTBA_i > T'(\sum_{i=1 \ldots n} \lceil \frac{T_{p_i}}{T'} \rceil) + T'$, where T' is the least common multiple of $T_{f_1}, T_{f_2}, \ldots T_{f_m}$, then we change $LTBA_i$ to $T'(\sum_{i=1 \ldots n} \lceil \frac{T_{p_i}}{T'} \rceil) + T'$. Both systems S and S' are equivalent regarding their behavior concerning deadline violation.*

Proof: The main difference to the previous case is that periodic functions with different tasks might influence each other. Potential decoupling points occur only at points in time, where all tasks start together. The proof idea is to view multiple periodic functions as a whole by taking the least common multiple. Here we omit the detailed proof. ∎

V. IMPLEMENTATION

For implementation, we extend the functionality of FTOS-Verify to test the applicability. The verification model is constructed in a format acceptable by UPPAAL [4]. Note that templates in UPPAAL are not completely suitable for our usage, since they only represent a fixed behavior with configurable parameters. Therefore, algorithms to automatically generate timed automata based on FTOS models are needed. We have implemented our automated M2M transformation tool using openArchitectureWare[9] under the Eclipse modeling framework[10].

As use case, we apply the verification in the context of our balanced-rod example[11], where the control functions are replicated on three redundant machines to guarantee fault-tolerance. All components mentioned previously are generated by our automatic conversion technique; the resulting UPPAAL system has 25 communicating timed automata. As timing information for the different components, we use currently user-specified assumptions. An integration of WCET-analyzers is foreseen. One desired property specified is the guarantee for the absence of deadline violation, which turns to be the reachability property in UPPAAL.

The overall execution time varies from 1 to 25 minutes depending on the accuracy of the verification model on a Intel 2.33 GHz machine using a FIFO-based priority-driven scheduler. The memory consumption can reach up to 850Mb. The verification of using a Round-Robin based scheduler showed to be too memory comsuming.

VI. RELATED WORK

We mention related work, but constrain ourselves in works regarding the analysis of Giotto-like systems; for techniques applying formal verification in real-time analysis, we refer readers to the survey paper by Wang [13]. In Giotto, the Giotto-Compiler will perform hardware mapping and apply analysis techniques to check schedulability. Many design tools with Giotto-like MoCs apply similar approaches, for example, TDL [11] or HTL [7], but analysis techniques are not explicitly mentioned. One interesting work comes from COMDES-II project [9], which is also based on the concept of logical execution time; here, researchers apply model transformation from system models to verification models. Nevertheless, as we focus on fault-tolerant systems, our work differs from the above works with the following facts. First, we encounter a harder problem; by applying software fault-tolerance, modeling the communication between multiple deployed units is required, and this is not required by other Giotto-like MoCs. For those MoCs, scheduling analysis developed in real-time community could be enough without the use of model checking. Furthermore, by proposing the

---

[9] http://www.openarchitectureware.org

[10] http://www.eclipse.org/modeling/emf/

[11] For configurations, see http://www6.in.tum.de for details.

similarity between aperiodic events and fault occurrences, our theoretical criteria is powerful to reduce dramatically the complexity of the model (not the verification algorithm). This is based on our understanding regarding constituents for the complexity of timed verification.

VII. CONCLUSION

In this paper, we discussed the issue of constructing a model to verify timing assumptions in the context of FTOS using timed automata. However, due to our general approach, the results can be applied to arbitrary distributed real-time systems.

Our contribution can be summarized as follows.

1) We give observations concerning modeling of general distributed real-time systems using timed automata and formulate our verification model construction process.

2) With the context of systems consisting of periodic and aperiodic tasks, we give theoretical criteria how to reduce the size of the verification model, which is particularly useful for our approach. The change of the maximum integer used in the system decreases the required time for verification with exponential scale.

3) A prototype software for the conversion process is constructed with preliminary experiments.

Our work is currently based on user-specified assumptions regarding the timing of involved components. The next step will be the integration of WCET analyzing tools to have a faithful verification result.

Furthermore, we are investigating on approaches to separate the verification problem for control functions executed in parallel to make our approach applicable also for large-scale applications.

REFERENCES

[1] Eclipse Modeling Framework. *http://www.eclipse.org/modeling/emf/*.

[2] openArchitectureWare Project. *http://www.openarchitectureware.org/*.

[3] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[5] Christian Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. PhD thesis, Technische Universität München, Oct 2008.

[6] C. Cheng, C. Buckl, J. Esparza, and A. Knoll. FTOS-Verify: Analysis and Verification of Non-Functional Properties for Fault-Tolerant Systems. Technical report (arXiv:0905.3946), TU Munich, 2009.

[7] A. Ghosal, A. Sangiovanni-Vincentelli, C.M. Kirsch, T.A. Henzinger, and D. Iercan. A hierarchical coordination language for interacting real-time tasks. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT'06)*, pages 132–141. ACM New York, NY, USA, 2006.

[8] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT'01)*, pages 166–184. Springer-Verlag, 2001.

[9] X. Ke, P. Pettersson, K. Sierszecki, and C. Angelov. Verification of COMDES-II Systems Using UPPAAL with Model Transformation. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 153–160. IEEE Computer Society Washington, DC, USA, 2008.

[10] A. Sangiovanni-Vincentelli and G. Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers*, pages 23–33, 2001.

[11] R. Simmons and D. Apfelbaum. A task description language for robot control. volume 3, pages 1931–1937 vol.3, Oct 1998.

[12] W. Torres-Pomales. Software fault tolerance: A tutorial. *Langley research Center, Hampton, Virginia, Technical Report, No. NASA/TM-2000-210616*, 2000.

[13] F. Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, 2004.

[14] F. Wang and C.H. Cheng. Program Repair Suggestions from Graphical State-Transition Specifications. In *Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems (FORTE'08)*, volume 5048 of *LNCS*, pages 185–200. Springer–Verlag, 2008.