DEPENDABILITY DESIGN FOR DISTRIBUTED REAL-TIME SYSTEMS WITH BROADCAST COMMUNICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YUSUF BORA KARTAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JUNE 2014

Approval of the thesis:

DEPENDABILITY DESIGN FOR DISTRIBUTED REAL-TIME SYSTEMS WITH BROADCAST COMMUNICATION

submitted by **YUSUF BORA KARTAL** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Engineering** ________________

Assoc. Prof. Dr. Şenan Ece Schmidt
Supervisor, **Electrical and Electronics Engineering Dept., METU** ________________

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU ________________

Assoc. Prof. Dr. Şenan Ece Schmidt
Electrical and Electronics Engineering Dept., METU ________________

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU ________________

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU ________________

Assist. Prof. Dr. Reza Hassanpour
Computer Engineering Dept., Çankaya University ________________

Date: **17.06.2014**

ABSTRACT

DEPENDABILITY DESIGN FOR DISTRIBUTED REAL-TIME SYSTEMS WITH BROADCAST COMMUNICATION

KARTAL, Yusuf Bora
Ph.D., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Şenan Ece Schmidt

June 2014, 98 pages

The operation of distributed systems relies on the timely exchange of message data via dependable communication networks. Previous works suggest hardware redundancy for potential faults in the underlying network infrastructure to achieve dependability. However, software faults and faults that cannot be resolved on the hardware level are not considered in the existing literature. This work proposes a new method for software fault-tolerant communication in distributed real-time systems with communication networks that support time-slotted operation and broadcast transmission.

Our method implements a dependability plane to be integrated to the existing network stack. It processes dependability information that is piggybacked on application message and uses a time synchronized checkpointing/rollback recovery strategy. The proposed dependability plane is modeled in the framework of timed input/output automata (TIOA) to formally prove its correctness and determine tight bounds for fault-recovery times. Model checking tools are employed to verify the timing and dependability properties of real-time systems. To this end, we present an algorithmic approach for converting TIOA models to be used as input of a well-known model checking software tool UPPAL. We apply our dependability plane design and integrate it to a previously developed real-time communications framework. We further verify the TIOA models of the overall protocol stack by employing our algorithmic conversion to UPPAAL.

Keywords: Distributed systems, real-time, communication, dependability, software fault-tolerance, timed input/output automata, UPPAAL, formal verification

ÖZ

YAYGIN HABERLEŞME YAPAN GERÇEK ZAMANLI DAĞITIK SİSTEMLER İÇİN GÜVENİLİRLİK TASARIMI

KARTAL, Yusuf Bora
Doktora, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Şenan Ece Schmidt

Haziran 2014 , 98 sayfa

Dağıtık sistemler güvenilir haberleşme ağlarına ihtiyaç duymaktadır. Her ne kadar güvenilirlik konusu donanım yedekleme şeklinde literatürde işlenmiş bir konu olsa da yazılım güvenilirliği ve donanım yedekleme ile giderilemeyecek hata durumları ele alınmamıştır.

Bu çalışmada, yaygın haberleşme yapan gerçek zamanlı dağıtık sistemlerde hata toleransı sağlanabilmesi için senkronize hata bloğu oluşturma ve geri dönmeye dayalı özgün bir güvenilirlik katman tasarımı önerilmektedir. Bahsi geçen güvenilirlik katmanı Zaman Girişli/Çıkışlı Otomat sentaksı kullanılarak modellenmiştir. Bu sayede hem işlevsel doğruluğunun formal yollarla kanıtlanması sağlanmış hem de hata kurtarma işlemindeki gecikmeler sıkı zaman kısıtları dahilinde öngörülebilmiştir. Zaman Girişli/ Çıkışlı Otomat (TIOA) işçerçevesi dağıtık sistemlerin modellenmesinde, UPPAAL yazılım paketi ise sistem modellemesi, simülasyonu ve doğrulamasında sıklıkla kullanılan araçlardır. Bu çalışmada, TIOA sentaksında modellenen güvenilirlik katmanının UPPAAL ortamına aktarılması için algoritmik bir yöntem geliştirilmiş ve çevrim yapılabilmesi için TIOA modellerinde bulunması gereken özellikler listelenmiştir. Literatürdeki diğer çalışmalardan farklı olarak önerilen çevrim yöntemi gerçek-zamanlı dağıtık sistemler üzerinde kullanılabilmektedir. Çalışma kapsamında gerçek-zamanlı dağıtık ve güvenilir bir haberleşme iş çerçevesi (D3RIP) uygulama örneği olarak verilmektedir.

Anahtar Kelimeler: Dağıtık sistemler, gerçek-zamanlı, haberleşme, güvenilirlik, yazılım hata toleransı, zaman girişli/çıkışlı otomat, UPPAAL, formal doğrulama

To my children Masal Ada and Mehmet Tuna

ACKNOWLEDGMENTS

I would like to express my special thanks to my supervisor Assoc. Prof. Dr. Şenan Ece Schmidt. I would like to thank to Assoc. Prof. Dr. Klaus Schmidt for his support. My special thanks go to the thesis jury members for their expert feedbacks. I would like to thank TÜBİTAK who supported this study as a scientific project (Career Award 109E261). Finally, I would like to thank to my wife and parents who supported me throughout the whole time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

AT	Acceptance Test
CL	Coordination Layer
DH	Dependability Header
DP	Dependability Plane
IL	Interface Layer
NRT	Non Real-Time
RT	Real-Time
SC	Synchronized Checkpointing
SDEP	Synchronization Based Dependability Protocol
SM	Shared Medium
TCOZ	Timed Communicating Object-Z
TCTL	Timed Computation Tree Logic
TDMA	Time Division Multiple Access
TIOA	Timed Input/Output Automata
TSIL	Time Slotted Interface Layer
UML	Unified Modelling Language
URT	Urgency Based Real-Time Protocol

CHAPTER 1

INTRODUCTION

Distributed real-time systems are widely used in areas such as factory automation, process automation, building automation or automotive control [3, 4, 5]. The distributed applications in such systems are dependent on the real-time data exchange among the distributed system components over a *communication network*. To this end, different network architectures with bus-based, switched or wireless communication facilities are adopted. Hereby, the dependability attributes of the communication network such as *maintainability* and *availability* [1] are key issues in order to support the timely delivery of communication messages for safety-critical real-time (RT) applications as well as the reliable support of non-real-time (nRT) traffic. A well-accepted method for increasing the dependability of communication systems is the addition of *hardware redundancy* to the transmission medium where the system switches to a redundant hardware resource in case a resource failure is encountered. However, *software faults* such as corrupted memory or buffers, race conditions, operating system faults that occur in the software protocol stack can not be resolved by hardware redundancy.

The distributed real-time computing systems are designed to satisfy certain timing constraints in addition to the application level behavioral constraints. Such systems require the analysis and verification of the system properties at the design stage before implementation. To this end, timed input output automata (TIOA), as introduced in [6], constitute a viable mathematical framework for modeling and analyzing real-time systems. Furthermore, this framework is suitable for distributed systems as the timed behavior of a system can be represented by the *composition* of multiple TIOA representing its individual components. It is possible to check certain practical conditions for individual TIOA models and the overall system model that is obtained after composition. A further method to check if the design of a real-time distributed system logically fulfills its timing and behavioral specifications is *model checking*. Specifically, model checking techniques provide complete proofs of specifications fulfillment for a given behavioral model of the system. UPPAAL [7] is a well-known automatic model checking software tool that is based on the modeling framework of *timed automata* (TA). It can be observed that, on the one hand, the TIOA framework provides convenient models for distributed real-time systems and one can manually construct formal proofs for certain system properties. On the other hand, UPPAAL is a software tool for formal verification and employs behavioral TA models of the system components which again should be constructed manually.

The focus of this thesis is the usage of *software redundancy* to achieve dependability attributes for the communication of distributed real-time systems, which is facilitated over a time-slotted network with broadcast capability, by using formal modeling, analysis and verification techniques. The contributions of the thesis are as follows:

- We propose a *dependability plane* which constitutes a software layer that operates based on the principle of *checkpointing* and *rollback recovery* [8]. Our dependability plane detects software faults by means of a synchronized *acceptance test* (AT) on all network nodes. Our AT inspects dependability information that is piggybacked on application messages. If the AT fails, all nodes roll back to a non-faulty state upon reception of a rollback message of the currently transmitting node. We model the proposed dependability plane using the timed input/output automaton (TIOA) formal framework [9] and show that all faults are resolved with a pre-computed bounded *recovery time*. We identify potential fault scenarios that cannot be resolved by the underlying hardware and hence need to be addressed in the software stack. We then formulate the basic requirements of achieving the desired dependability attributes for the described model.
- We present an algorithm TUConvert, that generates UPPAAL TA models from distributed TIOA models. This algorithm achieves a complete workflow where the TIOA models for system components are automatically converted to UPPAAL templates and then formally verified by UPPAAL. To this end, we establish the basic conditions for TIOA models that are suitable for such conversion.
- We apply the proposed dependability plane design to a family of distributed industrial real-time protocols proposed in [10] to demonstrate its features. We then use the TU-Convert Algorithm to obtain the UPAAL TA models for the overall protocol family including the dependability plane and verify its correct operation.

The remainder of this dissertation is organized as follows: Chapter 2 gives the necessary background information for a good understanding of the thesis work. The background section describes the software dependability concepts, checkpointing and rollback mechanisms together with the TIOA framework and UPPAAL toolsuite. Our novel dependability plane design for distributed real-time systems is given in Chapter 3 with the proposed checkpointing and rollback recovery mechanism. We apply our dependability design on a family of distributed industrial real-time protocols in Chapter 4. Chapter 5 defines our novel TUConvert algorithm which is used to convert TIOA models into UPPAAL TA models. We produce the UPPAAL TA models of the family of distributed industrial real-time protocols by using the TUConvert tool. The UPPAAL TA models of the framework layers are used for the simulation and model checking based formal verification studies in Chapter 6. Finally the dissertation gives some concluding remarks in Chapter 7.

Two journal papers are prepared as a result of this thesis study. The first one defines the

TUConvert algorithm, whereas the second one defines our dependability design for distributed real time systems.

CHAPTER 2

BACKGROUND

This chapter describes the required background information for the thesis study. Dependable computing concepts together with the checkpointing and rollback mechanisms for software fault tolerance are defined in Section 2.1 and 2.2 respectively. On the other hand the TIOA syntax for formal system modeling is described in Section 2.3 whereas UPPAAL as a simulation and formal verification tool-suite is given in Section 2.4. Finally Section 2.5 defines the real-time communication networks and the dependability design methodologies that are employed in real-time communication.

2.1 Concepts of Dependable Computing

Dependability is considered as the ability to deliver service that can be justifiably trusted [1]. The formal treatment of dependability in the literature is based on the definition of *threats* – actions that can affect the system's ability to perform its service; *attributes* – quantities that measure the system's dependability; *enforcement techniques* – methods that are intended to improve the system's dependability [11].

Threats comprise *faults*, *errors* and *failures*. Here, faults/errors describe the deviation of a device operation/internal system state from what is expected, whereby errors are the consequence of faults. Faults and errors can cause failures, which describe the deviation of the external system behavior from the specified one [11]. The precise relation between the dependability threat is shown in Figure 2.1.

As shown in Figure 2.1 a fault is defined as a problem that causes a component level error in the system operation. The effects of the component faults are not seen at the user level until the faulty component is used within the correct system operation. The errors propagate to other system components via the component interfaces and cause system failures at user level causing the system to deviate from correct operation.

Dependability is measured by means of attributes such as *reliability*, *maintainability*, *availability* and *safety*. Reliability/maintainability describe the probability of correct service at time $t > 0$, assuming that correct service/incorrect service is provided at time $t = 0$. The notion of availability concerns the probability of correct service at time $t > 0$ with the only assumption that repair is initiated immediately whenever incorrect service is detected. Safety ensures that unacceptable failures do not occur in the presence of faults.

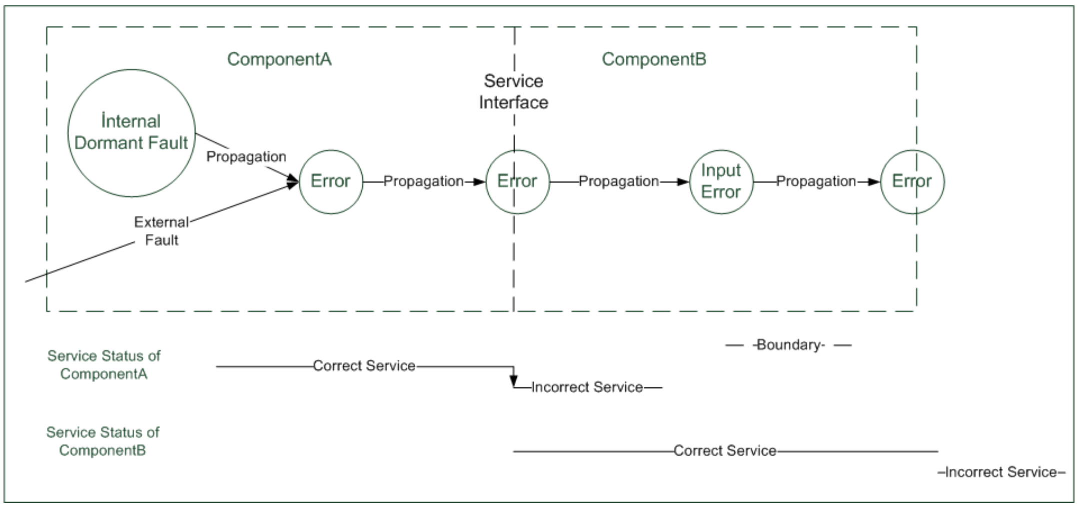


Figure 2.1: Dependability Threats [1]

Dependability of a system can be improved by using enforcement techniques such as *fault prevention*, *fault-tolerance*, *fault-removal* or *fault-forecasting* [1]. *Fault prevention* addresses the design level preventions to avoid the fault occurrences. The most effective design level fault prevention method is using structural design and development methodologies. *Fault tolerance* is described as the strength of a system to whitstand system faults. In order to achieve fault tolerance, the faults should be recognized and avoided before causing errors. *Fault removal* is the removal process of the detected faults via fault detection and identification. On the other hand *fault forecasting* is defined as the process of recognizing an incoming fault by looking at the the system state changes.

In this study, fault-tolerance for *fail-operational* systems is of particular interest. That is, it is desired to ensure continued correct service even in case of faults. Such task is generally achieved by inserting *predictive redundancy* into the system in order to counteract faults [11].

2.2 Checkpointing and Rollback Mechanisms for Software Fault Tolerance

Software fault-tolerance concerns faults that are not accounted by the system hardware or operating system [12]. A frequently used method for software fault-tolerance in distributed systems is given by the *checkpoint/rollback-recovery* strategy [13, 14]. The system state is recorded regularly on stable storage (checkpointing) and the system can hence recover its computation from such checkpoint in case of fault (rollback). Hereby, it has to be noted that information exchange among distributed processes requires communication through a network [8]. Moreover, considering distributed *real-time* systems, is further essential to achieve checkpoint/rollback-recovery without excessive computational effort and without impairing the real-time system behavior [15, 16].

Concurrent processes in a real-time system have the requirement of completing their execution within a pre-determined deadline. Failing to fulfill the timeline requirement can cause an error in the execution fragment. It is difficult to meet the deadline requirement if one or more processes in the system need restarts in case of a failure. Hence, it is a critical issue in the design of real-time systems to develop error recovery mechanisms which do not require restarts.

Since our main focus is achieving software fault tolerance for distributed real-time systems, the dependability requirements should not require restarts for error recovery. The recovery block approach is proposed in [17, 18] as means for error recovery without restart. In this approach, the distributed system entities save their states several times as a procurement for failure situations. The saved states are used to roll back in case of a failure situation. However this rollback scheme may result in a cascade of rollbacks which causes the system entities to roll back to the beginning of their executions. This problem is called as the *domino effect*. Domino effect is illustrated in Figure 2.2.

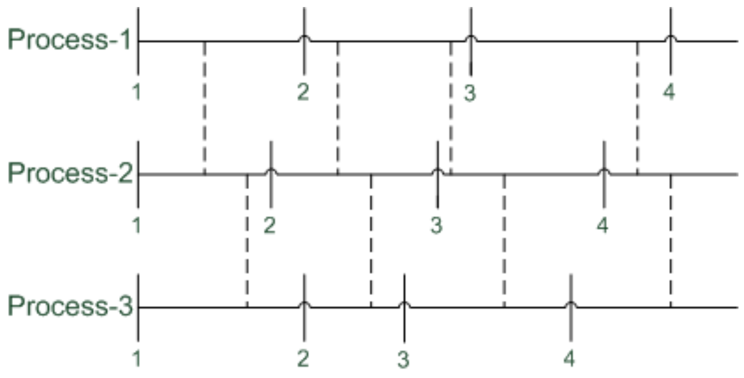


Figure 2.2: Domino Effect

The vertical dashed lines in Figure 2.2 show the inter-process communications. Suppose that a failure occurs in Process 3 after its 4th recovery block. This causes the third process to roll back to its 4th recovery block. Since Process 4 has interacted with other processes after its 4th recovery block, the other two processes should also rollback in order to recover their states before the 4th recovery block of Process 3. This chain continues and all the processes face with the obligatory situation to roll back to their initial states.

The literature offers three main approaches to checkpointing [8]. In *uncoordinated checkpointing*, distributed processes take checkpoints independently. Such approaches are not suitable for real-time systems since they can lead to the *domino effect* which causes a system restart. *Communication-induced checkpointing* enforces checkpointing based on information that is piggybacked on application messages. That is, checkpointing is coordinated among distributed processes without introducing additional messages [19, 20]. In contrast, *coordinated checkpointing* potentially relies on excessive message passing and computations in order to coordinate checkpointing among distributed processes [8]. However, if clock synchronization among distributed processes is available, checkpointing can be coordinated without further message exchange [21, 22, 15].

A particular method for checkpointing/rollback recovery is based on *recovery blocks* and *acceptance tests* (AT) [17, 18]. It assumes an application that is structured in a *primary block* that is followed by an AT. If the AT fails, the application enters an *alternate block* that has to perform the desired operation. After a further AT, the application continues its normal operation. In this study, we develop a checkpointing/rollback recovery method for distributed real-time systems that is based on coordinated checkpointing with synchronized clocks and a particular recovery block.

2.3 Timed Input/Output Automata

Correctness or performance of timed systems depends on the timing of events and the order of their occurrence as well as the system evolution between events [2]. The timed input/output automata (TIOA) framework is developed as a general mathematical framework for modeling and analyzing timed systems. The fundamental object in the TIOA framework is the *timed input/output automaton*. TIOA framework enables the decomposition of a system which eases the analysis of a distributed system. In this section, we point out the relevant properties and features of TIOA for our conversion method.

According to the definition in [2], a TIOA is represented by a 6-tuple $\mathcal{A} = (X, Q, Q_0, S, D, T)$ with the following entries:

- X: set of *variables* that are internal to $\mathcal{A}$,
- Q: set of *states* with $Q \subseteq val(X)$. Here $val(X)$ denotes the possible valuations of X,
- $Q_0 \subseteq Q$: set of *initial states* with $Q_0 \neq \emptyset$,
- S: *signature*, which lists the TIOA actions together with their types. There are *input actions* I, *output actions* O and *internal actions* H,
- D: set of discrete *transitions*. Each transition is considered as a triple $(q, a, q') \in Q \times S \times Q$,
- T: set of trajectories. Trajectories have a domain, which is a specific subset of the set of the real numbers. For each $\tau \in T$ and time instant $t \in \mathbb{R}$, $\tau(t) \in Q$. Here, $\tau(t)$ denotes the state valuation of the trajectory τ at time t.

In this definition, each variable $v \in X$ possesses a *static* and a *dynamic* type. The static type captures the possible range of values of v, whereas the dynamic type captures how the valuation of v evolves in time. If v is a *discrete variable*, its value can only change at discrete time instants, whereas an *analog variable* v can change continuously. For convenience, we write X^{d} for the discrete variables and X^{a} for the analog variables. Hence, the state Q of the TIOA either changes instantaneously with the occurrence of discrete transitions (if a related transition is possible) or continuously during the evolution of trajectories. An action $a \in S$ is enabled at a state $q \in Q$ if the related discrete transition is possible, that is there is a state $q' \in Q$ such that $(q, a, q') \in D$. According to the TIOA semantics, input actions ($a \in I$) are always enabled, whereas output or internal actions ($a \in O \cup H$) are enabled depending on D. It has to be noted that the transition relation D generally characterizes an infinite number of transitions. However, D usually need not be enumerated explicitly, but can be formulated in terms of *pre-conditions* on Q that enable actions and the *effect* of a transition as a rule to compute the new state after taking the transition.

In order to model physically relevant behavior, several practical conditions are introduced for TIOA. In particular, a TIOA has to be *input action enabling*, i.e. $q \in Q$ and $a \in I \Rightarrow \exists$ $q' \in Q$ s.t. $q \xrightarrow{a} q'$. Furthermore, a TIOA must be *time passage enabling*: $q \in Q \Rightarrow \exists\, \tau \in \mathcal{T}$ s.t. $\tau.\mathit{fval} = q$ and either $\tau.\mathit{ltime} = \infty$ or $\tau.\mathit{ltime} < \infty$ and $\exists\, q' \in Q, l \in L$ s.t. $\tau.\mathit{lval} \xrightarrow{l} q'$. In this expression, $\tau.\mathit{fval}$, $\tau.\mathit{lval}$ and $\tau.\mathit{ltime}$ denote the first valuation, last valuation and last time of trajectory τ. Similarly, $\tau.\mathit{ftime}$ denotes the first time of trajectory τ. Each particular run of a system that is modeled by a TIOA $\mathcal{A}$ is described by an *execution*. In this context, an *execution fragment* is an (S, X)-sequence $\alpha = \tau_1 a_1 \tau_1 a_2 \cdots$, $\tau_i \in \mathcal{T}$ and $a_i \in S$, over the actions S and the variables X, whereby it holds for the trajectories $\tau_i \in \mathcal{T}$ and the actions $a_i \in S$ that $\tau_i.\mathrm{lval} \xrightarrow{a_{i+1}} \tau_{i+1}.\mathrm{fval}$. That is, α records all discrete and continuous state changes that happen during a system run. α is denoted as an *execution* if it starts from an initial state $\alpha.\mathrm{fval} \in Q_0$.

We next introduce a compact TIOA representation based on the TIOA language [23, 2]. In this representation, a TIOA is defined by its *header*, *state variables*, *signature*, discrete *transitions* and continuous *trajectories* with the following compact notation.

The **header** of a TIOA is given as the automaton name together with a list of parameters, that either represent fixed values of a certain type or that denote a type. In the first case, a parameter is for example written as n : *Real*, whereby n represents a value of type *Real*. In the second case, we would write $M : \mathit{Type}$ for a type M. Assuming that P_F and P_T represent a list of fixed and type parameters, respectively, the header is written as

$$\textbf{automaton}\ \mathit{Name}(P_F, P_T)$$

States of a timed input/output automaton are the valuations of the variables X. The static types of these variables are declared in the variables list in the form

$$x : T := x_0$$

for a variable $x \in X$ with type T and initial valuation x_0.

The **signature** of a TIOA is given as a list of TIOA actions. We represent each action by

$$\textbf{action type}\ \ \text{A}(y_{\text{A}1} : Type, \cdots, y_{\text{A}k_{\text{A}}} : Type).$$

That is, each action is described by its action type (input, output, internal), its name (A) and an optional list Y_{A} of k_{A} parameters $y_{\text{A}1}, \ldots, y_{\text{A}k_{\text{A}}}$ with their respective types.

Transitions of a TIOA are specified using a precondition (pre:) and an effect (eff:) in the form

internal A	**output** $\text{A}(Y_{\text{A}})$	**input** $\text{A}(Y_{\text{A}})$
pre: $f_{\text{A}}(val(X))$	pre: $f_{\text{A}}(val(X)) \wedge Y_a = u_{\text{A}}(val(X))$	pre:
eff: $val(X) := h_{\text{A}}(val(X))$	eff: $val(X) := h_{\text{A}}(val(X))$	eff: $val(X) := h_{\text{A}}(val(X), Y_{\text{A}})$.

Hereby, we distinguish internal, output and input transitions. $f_{\text{A}} : val(X) \to Bool$ is a predicate on the variable valuation that defines the enabling condition for the transition in case of internal and output transitions. There is no precondition for input transitions. The parameters Y_a are determined as $Y_a = u_{\text{A}}(val(X))$ when taking an output transition. The state update is performed as $h_{\text{A}}(val(X))$ in case of internal and output transitions. For input transitions, $h_{\text{A}}(val(X), Y_{\text{A}})$ performs the state update using the variables Y_{A} that are passed when taking the transition.

Trajectories of a TIOA are defined by the time evolution of analog variables as well as stop conditions, that determine when further advance of time is not allowed.

$$\textbf{stop when}\ \ w(val(X))$$
$$\textbf{evolve}\ \ d(x) \,\square\, e_x \text{ for } x \in X^{\text{a}}$$

Hereby, $d(x)$ represents the time derivative of x, $\square$ represents $=, \leq, \geq$ and e_x is a real-valued expression containing valuations in $val(X)$. It is assumed that the analog variables evolve according to continuous functions, whereby the discrete variable values remain constant throughout a trajectory. The stopping condition $w : val(X) \to Bool$ is fulfilled if the only state where $w(val(X))$ is true is the last state of the trajectory. In that case, either a discrete transition occurs, after which further time evolution is possible or the execution of the TIOA terminates.

In order to illustrate the previously introduced notation, TIOA examples are given in Fig. 2.3. The automata are taken from [2]with small modifications. Fig. 2.3 shows two TIOA with the names $UseOldInputA$ and $UseOldInputB$, which implement a peer-to-peer messaging scheme. $UseOldInputA$ receives the parameter $maxIt$ with type Int, whereas $UseOldInputB$ has no parameters. The automaton states are defined by the valuation of three variables. The discrete variable $\texttt{maxout}^{\text{d}}$ is used as a counter variable, limiting the number of messages. The analog variable $\texttt{now}^{\text{a}}$ captures the evolution of time, whereas the discrete variable $\texttt{next}^{\text{d}}$ is used to define the messaging instants. The signature comprises the actions A and B. A is used as a parametrized output action with parameter cnt in $UseOldInputA$ automaton and as a parametrized input action with parameter cnt in $UseOldInputB$.

Looking at the discrete transitions in $UseOldInputA$, A can happen if the precondition $\texttt{maxout}^{\text{d}} > 0 \wedge \texttt{now}^{\text{a}} = \texttt{next}^{\text{d}}$ is fulfilled. In that case, the discrete variables $\texttt{maxout}^{\text{d}}$

and $\texttt{next}^d$ are updated and the parameter $cnt = \texttt{maxout}^d$ is passed. If B happens, $\texttt{next}^d$ is updated. For the discrete transitions in *UseOldInputB*, it holds that the occurrence of A updates the discrete variables $\texttt{maxout}^d$ and $\texttt{next}^d$. The output transition B can happen if $\texttt{maxout}^d > 0 \wedge \texttt{now}^a = \texttt{next}^d$. In that case, $\texttt{next}^d$ is updated. In both TIOA, time evolves continuously with a rate of 1 and the time evolution stops when $\texttt{now}^a = \texttt{next}^d$. Hereby, note that the variables are local to the automata, meaning that each TIOA has an own instance of variables with the same name.

```
automaton UseOldInputA(maxIt : Int)

state variables
  maxoutA^d : Int := maxIt
  nowA^a : Real := 0
  nextA^d : double := 0
signature
  output A(cnt : Int)
  input B()

transitions
output A(cnt)
  pre:
    (maxoutA^d > 0)∧
    (nowA^a = nextA^d)∧
    cnt = maxoutA^d
  eff:
    maxoutA^d := maxoutA^d − 1;
    nextA^d := infty;
input B()
  eff:
    if nextA^d = infty
      nextA^d := nowA^a + 1

trajectories
stop when
  nowA^a = nextA^d
evolve
  d(nowA^a) = 1


automaton UseOldInputB()

states
  maxoutB^d : Int := 0
  nowB^a : Real := 0
  nextB^d : double := 0
signature
  input A(cnt : Int)
  output B()

transitions
output B()
  pre:
    (maxoutB^d > 0)∧
    (nowB^a = nextB^d)
  eff:
    nextB^d := infty;
input A(cnt)
  eff:
    maxoutB^d := cnt
    if nextB^d = infty
      nextB^d := nowB^a + 1

trajectories
stop when
  nowB^a = nextB^d
evolve
  d(nowB^a) = 1
```

Figure 2.3: TIOA for UseOldInputA and UseOldInputB [2]

The previous TIOA definition captures the behavior of a single TIOA. If distributed systems are considered, it is beneficial to model each system component by a separate TIOA. We next describe the interaction of multiple TIOA in the TIOA framework. The TIOA syntax defines the composition operation to form complex systems from individual TIOA. Consider two TIOA $\mathcal{A}_i = (X_i, Q_i, Q_{0,i}, S_i, D_i, T_i)$, $i = 1, 2$. $\mathcal{A}_1$ and $\mathcal{A}_2$ are called *compatible* if they do not share any variables, internal actions and output actions. That is, $X_1 \cap X_2 = \emptyset$, $H_1 \cap S_2 = H_2 \cap S_1 = \emptyset$ and $O_1 \cap O_2 = \emptyset$. The composition $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2$ of two compatible TIOA combines the variables of $\mathcal{A}_1$ and $\mathcal{A}_2$, and identifies the shared external actions of $\mathcal{A}_1$ and $\mathcal{A}_2$ by their common name. Formally, the composition is defined by

- $X = X_1 \cup X_2$,
- $Q = \{x \in val(X) | x \lceil X_i \in Q_i, i = 1, 2\}$, where $x \lceil X_i$ is the restriction of x to the variables in X_i for $i = 1, 2$,

- $Q_0 = \{x \in val(X) | x\lceil X_1 \in Q_{0,1} \wedge x\lceil X_2 \in Q_{0,2}\}$,
- $I = (I_1 \cup I_2) - (O_1 \cup O_2)$, $O = O_1 \cup O_2$, $H = H_1 \cup H_2$, $S = I \cup O \cup H$,
- for $x, x' \in Q$ and $a \in S$, $x \xrightarrow{a} x'$ iff for $i = 1, 2$ either $a \in S_i$ and $x\lceil X_i \xrightarrow{a} x'\lceil X_i$ or $a \notin S_i$ and $x\lceil X_i = x'\lceil X_i$,
- $T = \{\tau \in trajs(X) | \tau \downarrow X_1 \in T_1 \wedge \tau \downarrow X_2 \in T_2\}$, where $\tau \downarrow X_i$ restricts the trajectory τ to the variables in X_i for $i = 1, 2$. $trajs(X)$ is the set of all trajectories over X.

In principle, both TIOA in the composition evolve independently. Only if one of the TIOA performs an output transition involving a shared external action a, the input transitions in the other automaton with the same name a take place synchronously.

An important property of a timed I/O automaton is the *progressiveness*, stating that the automaton has no *locally-Zeno* behavior [2]. In other words progressiveness of an automaton implies that the automaton can never make an infinite number of state transitions within a finite time interval. The *composition* operation of timed I/O automata is *associative* and preserves progressiveness.

Theorem 1 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two progressive TIOA. Then, $\mathcal{A}_1 || \mathcal{A}_2$ is also progressive [2].*

The composition of *UseOldInputA* and *UseOldInputB* in Fig. 2.3 has the output actions A and B, whereas there are no input actions. The initial state is system system has a unique initial state with $\mathtt{maxoutA}^{d} = \mathtt{maxoutB}^{d} = maxIt$, $\mathtt{nowA}^{a} = \mathtt{nowB}^{a} = 0$ and $\mathtt{nextA}^{d} = \mathtt{nextB}^{d} = 0$. Time evolution of the composed system is represented by the real-valued variables $\mathtt{nowA}^{a}$ and nowB^{a} which increase with a rate of 1. Since *UseOldInputA* and *UseOldInputB* in Fig. 2.3 are progressive, their composition *UseOldInputA* || *UseOldInputB* is also progressive.

2.4 UPPAAL as a Formal Modeling and Verification Environment

Formal approaches are frequently employed to check whether a system design logically fulfills its timing and behavioral specifications [24, 25]. Model-checking is a powerful technique that can provide complete proofs of specifications fulfillment if the system behavioral model is given as a concurrent state-graph model. Model checking is usually carried out by comparing the behavioral model of a system with a temporal logic formula defining the system level requirements. Short verification time, reduced complexity with the possibility of partial function checking and the availability of sequential system behavior verification are the advantages of model-checking in the formal verification process [26, 27]. UPPAAL [7] is a well-known automatic verification software tool. It has been used to verify the operation of different real-time systems such as a robot scrub nurse [28], POSIX operating systems [29] and timed multitask systems [30]. UPPAAL uses timed automata extended with data variables as the description language. It also has a simulator that is used to see the behavior of the system.

UPPAAL is a tool suite for modeling, simulation and verification of real-time systems [7, 31] based on the theory of timed automata (TA) [32]. Typical applications of UPPAAL are control applications and communication protocols where timing plays a critical role.

UPPAAL consists of three main parts: a description language, a simulator and a model-checker. UPPAAL uses a non-deterministic description language extended with data types [31] [33]. The simulator is a validation tool that allows examining the functional behavior of the modeled system. The simulator provides an inexpensive mean for the detection of faults and inconsistencies in the early design stage. There are two different simulators in UPPAAL. The symbolic simulator (simulator) is used to examine the possible dynamic executions (symbolic traces) of a system as a means for validation. The concrete simulator on the other hand is used to examine the concrete traces of a system where the specific time to fire a transition can be selected. The verifier is used to check safety and liveness properties by exploring the state-space of the system. The verifier provides a requirement specification editor for specifying the system requirements in terms of TCTL (Timed Computation Tree Logic) queries. The basic workflow of UPPAAL is given in Fig. 2.4.

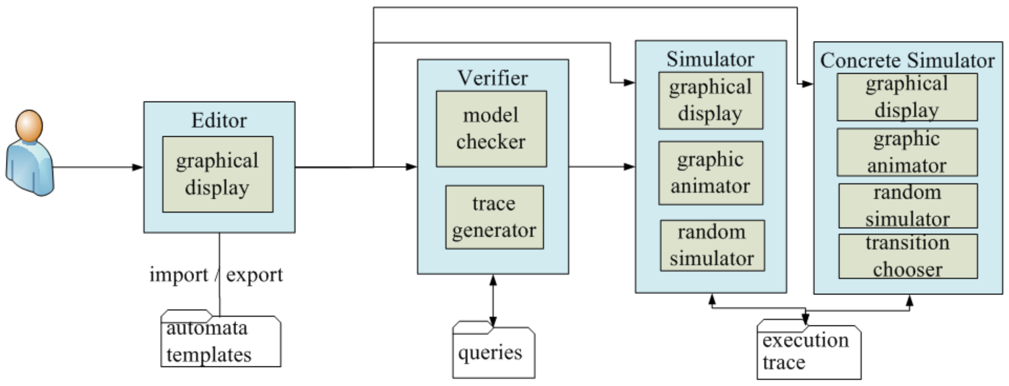


Figure 2.4: UPPAAL Operation Overview

As shown in Fig. 2.4, UPPAAL operation is started by an operator with a graphical model of the system entities. The input model is used as a *template* in order to create behavioral instances. Input UPPAAL templates depend on a timed automaton extended with certain data structures such as bounded integer variables, templates and constants. UPPAAL templates can be important from other system models.

Systems, composed of multiple automata, can be defined by creating the instances of the input templates. Both symbolic and concrete simulations can be run over the defined systems. UPPAAL uses the verification engine *Verifyta* to verify the systems. The operator writes a number of verification queries in order to check if the model requirements are satisfied. The queries for model-checking are written in text based Timed Computation Tree Logic (TCTL) syntax [34] [35]. UPPAAL's verifyta engine uses these text based queries as inputs. In addition, the verifyta engine is capable of generating symbolic traces for the unsatisfied queries. This feature makes the diagnostic analysis of a system model easier via showing the violating state transition for a specific query.

Example applications of UPPAAL for a communication channel and a real-time scheduling framework are given in [36] and [29] respectively.

In order to represent TA as defined in UPPAAL, we refer to the notation in [31, 37]. We employ the tuple $\mathcal{U} = (F, L, l_0, V, v_0, C, A, I, E)$ where:

- F is the set of parameters,
- L is the set of locations with the initial location $l_0 \in L$,
- V is the set of variables with the initial valuation v_0,
- C is the set of channels with input channel $c?$ and output channel $c!$ for each $c \in C$,
- $A = \{c!, c? | c \in C\} \cup \{\tau\}$ is the set of actions with the silent action τ,
- $I : L \times val(V) \rightarrow Bool$ assigns an invariant for the variable valuation to each location $l \in L$
- $E \subseteq L \times (val(V) \rightarrow Bool) \times A \times (val(V) \rightarrow val(V)) \times L$ is a set of edges.

We now give a more detailed explanation of the TA definition. F represents parameters that can be different for different instances of a TA. These parameters are substituted in the TA declaration in UPPAAL. A TA is permitted to stay in a location $l \in L$ as long as its invariant $I(l, v)$ evaluates to true for $v \in val(V)$. If the invariant is violated, either a discrete edge has to be fired from l or time has to stop which leads to an invalid execution of the TA. Since invariants are directly associated to locations, we use the notation l.inv to refer to the invariant of location l. In UPPAAL, an invariant is defined as a conjunction of boolean expressions which comprise simple conditions on clocks, differences between clocks and boolean expressions without clocks. Clock bounds should be given by integer variables and lower bounds on clocks are disallowed [38] .

Variables are divided into *local variables M* and *global variables G* such that $V = M \cup G$. Hereby, the valuation of local variables in M can only be updated by the TA itself, whereas the valuation of global variables in G can also be updated by the environment. Moreover, a subset of the variables constitutes clock variables that evaluate to a real number and evolve with a time derivative of 1 or 0. In the latter case, the clock realizes a stopwatch. We write K for the set of clocks and assume that all remaining variables are discrete variables that can only change when an edge is fired.

Channels synchronize the firing of edges in different TA: in the case of a binary synchronization channel $c \in C$, an edge with output channel $c!$ synchronizes with one edge with input channel $c?$; in the case of a broadcast channel $c \in C$, an edge with output channel $c!$ synchronizes with all edges with input channel $c?$ that can participate in the synchronization.

Actions consist of output actions of the form $c!$, input actions of the form $c?$ and the internal action τ that is not synchronized. For each edge $e = (l, g, a, r, l') \in E$, $g : val(V) \rightarrow Bool$ is a predicate that denotes the *guard*, a is an action and $r : val(V) \rightarrow val(V)$ denotes the *update* function. If e fires at a state $v \in val(V)$, the state is updated to $r(v)$. Hereby, the edge e can fire if and only if g evaluates to true for the current variable valuation and $I(r(v)) = true$. We write e.guard for the guard and e.update for the update function of e. Moreover, we specify the synchronization property of e using the related channel $c \in C$ and channel type ($c?$ or $c!$) by e.sync. If the firing of e is silent, e.sync is unused.

UPPAAL templates that realize the same behavior with the timed I/O automata $UseOldInputA$ and $UseOldInputB$ are given in Fig. 2.5. Both templates have a single initial location *l_init*. The initial location is tagged as committed in order to guarantee that the operation within the template is started with the outgoing edge transition from the initial location. Both automata define the same variables. The discrete variable *maxout* limits the number of discrete transitions whereas *next* is the discrete variable defining the time instants to fire an edge with an output channel. The analog variable *now* is used to keep track of the time evolution. The automata make their variable initializations while taking the first edge transition from the initial location *l_init* to the idle location *l_idle*. *l_idle* location is the second location for each automaton. Time passage is allowed within that location until one of the edges with the actions *a?*, *b?* (input channel) or *a!*, *b!* (output channel) is fired.

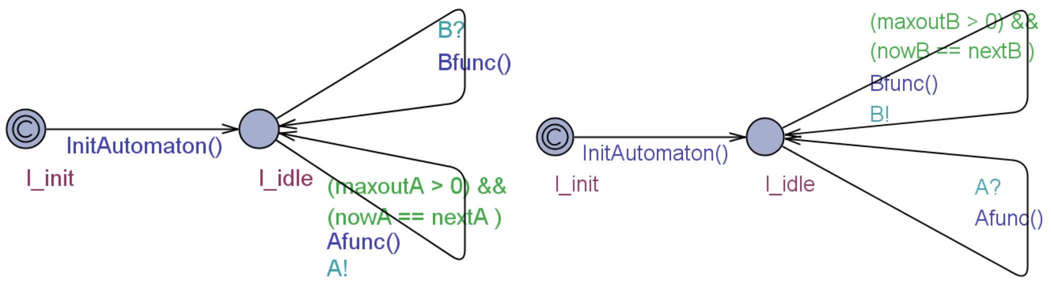


Figure 2.5: UPPAAL Templates for UseOldInputA and UseOldInputB

Regarding the composition of both automata, the operation is started with the action *a!* when the corresponding guard condition *(maxout > 0) && (now == next)* is fulfilled. $UseOldInputB$ synchronizes with the action *a?* and sets it local variables according to the global variable *cnt*. The function *afunc()* implements the *effect* field of the input transition *a?*. $UseOldInputB$ automaton continues the operation with the action *b!*. $UseOldInputA$ synchronizes with *b?* action and updates its local variables according to its internal state within the function *bfunc()*. The operation is continued until *maxout* reaches the limit *maxIt* which is passed as a parameter to $UseOldInputA$.

2.5 Real-time Communication Networks and Dependability

Many real-time embedded systems such as contemporary automation and manufacturing systems are implemented as distributed systems to realize complicated tasks where the participating devices are required to coordinate their operation. To this end, communication networks are employed to transport the messages that are generated by the system components.

These networks are required to guarantee the *timely delivery* of messages. In particular, real-time (RT) messages that are either *periodic* (e.g., from position control) or *sporadic* (e.g., from limit switches) have to be delivered with small delays. In addition to RT messages, non-real-time (nRT) messages related to time-uncritical tasks [39] and data communications for diagnosis or maintenance [40, 41] should also be delivered.

In the recent years, generic network technologies that are widely used for home and office networks and can be implemented with inexpensive COTS components are employed for real-time communication. To this end, *Ethernet* is the most promising standard [5]. Furthermore, there are efforts to employ wireless standards such as IEEE 802.15.1/BT, IEEE 802.15.4/ZigBee and IEEE 802.11/WLAN [42]. Many of these standards such as shared-medium Ethernet (IEEE 802.3) and wireless standards are broadcast networks with timing and synchronization support. Despite their advantages, these standards do not inherently support the timely delivery requirements of the industrial applications. Hence, it is an important research problem to achieve real time guarantees over these standards without modifying the network interface hardware to ensure the compatibility and low cost implementation.

It is critical to provide dependability support for such communication networks to support the timely delivery of communication messages [41, 5]. A well-accepted method for increasing the dependability of communication systems is the addition of *hardware redundancy* to the transmission medium [43, 44, 5]. That is, the system switches to a redundant resource in case a resource failure is encountered. Here, the network, network interfaces and entire network devices can be considered as resources [45, 46]. The academic literature provides various approaches for network redundancy based on fast recovery of switches [47, 48] and standardized solutions such as the media redundancy protocol [44]. In addition, different industrial Ethernet protocols are equipped with redundant network devices (such as Ethernet Powerlink [49], FTT-Ethernet [50]), redundant network interfaces (such as EtherCAT [51], Ethernet for Plant Automation [52]) and redundant network channels (such as SERCOS III [53], Time Critical Control Network [54]).

Although the cited methods support dependability with respect to faults in the network hardware, they do not consider *software faults* such as corrupted memory or buffers, race conditions, operating system faults that occur in the software protocol stack and can hence not be resolved by hardware redundancy. A notable difference is the work in [55] that uses redundant controller tasks in order to resolve faults such as corrupted message receptions on TDMA-based protocols such as FlexRay.

CHAPTER 3

DEPENDABILITY DESIGN FOR DISTRIBUTED REAL-TIME SYSTEMS WITH BROADCAST COMMUNICATION

This chapter defines a general software dependability mechanism for distributed-real time systems using broadcast communication. We first define the general properties of considered RT systems in Section 3.1. Then we develop our distributed rollback strategy depending on synchronized checkpointing in Section 3.2.

3.1 Distributed Real-Time System Model

Before giving our dependability design, we first define the general properties of distributed real-time systems under discussion. Section 3.1.1 lists the general properties of considered distributed real-time systems. Then Section 3.1.2 defines a general node model representing the distributed real-time system node in the framework of TIOA. We define potential fault scenarios for the distributed system models in Section 3.1.3. Finally we list certain dependability requirements of the systems under discussion in Section 3.1.4.

3.1.1 General Properties of Distributed Real-Time Systems

This section defines the general properties that a distributed real-time system should have in order to apply our dependability design. We first give a basic system architecture for a distributed real-time system in Fig. 3.1. The system is composed of distributed nodes communicating over shared medium. That is each transmitted message in the network is received by all nodes. Each process running on distributed nodes sends and receives messages regularly. Hence each node in the network have a frequent medium access.

The list of properties that the distributed real-time system should have is as follows:

P1 Synchronization: The distributed system entities (nodes) should operate in a time synchronized fashion.

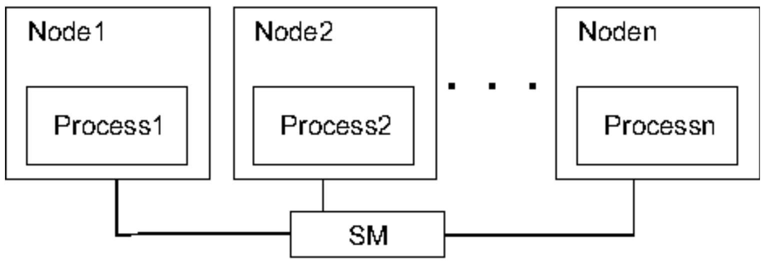


Figure 3.1: System architecture.

P2 Cyclic Operation: The distributed nodes should communicate in a cyclic manner. It is better to have a time slotted communication where each time-slot is assigned to a unique node.

P3 Broadcast Communication: Each transmitted message in the communication infrastructure should be received by all nodes in the network.

P4 Regular Network Access Per Node: In order to get the state information regularly, each node should send a message within a limited time which can be calculated before system operation.

P5 Knowledge About TX Instants of Each Node: Each node in the network should know the message transmission instants of the other nodes, so that it can recognize an untransmitted or wrong transmitted message.

P6 Regular Update of State Information: In order to have an updated state information for consistency checks, each node should update its state information regularly after each message reception.

P7 Consistent State Information Among Nodes: There should be a state information which is consistent at each node for certain time instants in order to make consistency checks in a distributed manner.

There are certain synchronization procedures such as IEEE1588 [56] to obtain distributed synchronization defined in P1. The exist system examples communicating in a ring topology on a shared medium having the properties P2 and P3. The systems composed of sensors and actuators are the example system architectures where each node has a regular netowrk acsess with updated state information (properties P4 and P6). Systems implementing distributed resource allocation have consistent state information (P7) and each node knows about the message transmission instants of other nodes (P5). Hence there are systems that have the listed properties P1-P7.

3.1.2 Generic Node Model

In order to formalize the discussion, we next develop a TIOA model of a generic network node that fulfills the properties stated in Section 3.1.1. To this end, we first reason about the event timing in such generic node according to Fig. 3.2. In principle, P1 and P2 imply that there is a common slot time T that captures the duration of one time slot of the cyclic system operation. Considering P3 to P6, the possible actions taken by each node are transmitting (TX) and receiving messages (RX) as well as data processing (PROCESS). We assume that PROCESS is performed in any time slot, whereas the occurrence of TX and RX depends on the system state. Accordingly, we denote a slot as *transmission slot* if all actions are executed. Otherwise the slot is a *spare slot*. Considering P6, the temporal order of these actions in a transmission slot is as follows. First, one of the nodes (P4) transmits a message if available and all nodes receive this message (P3). Respecting P1, we assume that transmission is always started before time $t_1 \geq 0$ and reception is completed before t_2, whereby $t_1 < t_2 < T$. After that, data processing is performed between t_2 and T before the end of the time slot. In spare slots, only PROCESS occurs between time t_2 and T.

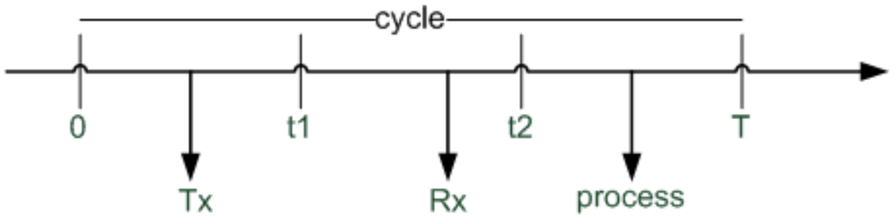


Figure 3.2: Timing diagram of the RT system operation.

We next present the TIOA model in Fig. 3.3 that captures the described timing behavior of a generic node i. Time is represented by the analog variable $\mathtt{now}_i^{\mathrm{a}}$ that evolves with time derivative $d(\mathtt{now}_i^{\mathrm{a}}) = 1$. The local state of the node is stored in $\mathtt{ST}_i^{\mathrm{d}}$ and $\mathtt{Tx}_i^{\mathrm{d}}$ and $\mathtt{Rx}_i^{\mathrm{d}}$ represent the transmit and receive message buffers respectively. $\mathtt{ST}_i^{\mathrm{d}}$ and $\mathtt{Tx}_i^{\mathrm{d}}$ are initialized by I_{ST} and I_{m}, respectively, whereas $\mathtt{Rx}_i^{\mathrm{d}}$ is initially empty. As indicated in Fig. 3.2, the possible actions are TX$_i$ (output), RX (input) and PROCESS$_i$ (internal). TX$_i$ is possible if time is smaller than t_1 ($\mathtt{now}_i^{\mathrm{a}} \leq t_1$), a message is ready in the Tx buffer ($\neg\mathtt{Tx}_i$.empty()) and the current time slot is allocated to the node for message transmission ($f_{\mathrm{oc}}(ST) = i$). In transmission slots, the function $f_{\mathrm{oc}} : A_{\mathrm{ST}} \rightarrow \mathbb{N}$ returns a unique node for transmission based on the state variable valuation in agreement with P4 and P5. In particular, we assume that there is an interval of Δ_{I} time slots that indicates the largest number of time slots between two transmissions of each node. In spare slots, f_{oc} returns 0 such that no node transmits a message. Note that it is ensured by the stop condition $\mathtt{now}_i^{\mathrm{a}} = t_1 \wedge f_{\mathrm{oc}}(\mathtt{ST}_i^{\mathrm{d}}) = i \wedge \neg\mathtt{Tx}_i^{\mathrm{d}}$.empty() that TX$_i$ always occurs before t_1. If TX$_i$ occurs, the content of the transmit buffer is sent ($m := \mathtt{Tx}_i$) and removed from the buffer (set $\mathtt{Tx}_i$ empty). Messages received by RX are stored in the receive buffer ($\mathtt{Rx}_i^{\mathrm{d}} := m$). Data processing is performed after t_2 and before T ($t_2 \leq \mathtt{now}_i^{\mathrm{a}} \leq T$). If PROCESS$_i$ happens, the state and transmit buffer are updated ($\mathtt{ST}_i^{\mathrm{d}} := f_{\mathrm{ST}}(\mathtt{ST}_i^{\mathrm{d}}, m)$ and $\mathtt{Tx}_i^{\mathrm{d}} := f_{\mathrm{Tx}}((\mathtt{ST}_i^{\mathrm{d}}, m))$, the slot time is reset ($\mathtt{now}_i^{\mathrm{a}} := 0$) and the receive buffer is emptied (set $\mathtt{Rx}_i^{\mathrm{d}}$ empty). Hereby, the function $f_{\mathrm{ST}} : A_{\mathrm{ST}} \times M \to \{0, 1\}$ performs the state update based on the state variables and $f_{\mathrm{Tx}} : A_{\mathrm{ST}} \times M \to M$ determines new messages for transmission.

The variables $\mathtt{mess}^{\mathrm{d}}$, $\mathtt{next}^{\mathrm{d}}$, $\mathtt{coll}^{\mathrm{d}}$ and $\mathtt{now}^{\mathrm{a}}$ define the variables of the TIOA. $\mathtt{mess}^{\mathrm{d}}$ (type M) holds the current message on the network, $\mathtt{next}^{\mathrm{d}}$ (type int) represents the completion of the next message transmission, $\mathtt{coll}^{\mathrm{d}}$ (type bool) is true if there is a collision – two messages from different nodes at the same time – on the network. $\mathtt{now}^{\mathrm{a}}$ (type Real) captures the time after the last message transmission.

```
TIOA $Node_i(t_1 : int, t_2 : int, T : int, A_{\mathrm{ST}} : Type, I_{\mathrm{ST}} : A_{\mathrm{ST}}, M : Type, I_m : M)$

states
  $\mathtt{now}_i^{\mathrm{a}} : R := 0$
  $\mathtt{ST}_i^{\mathrm{d}} : A_{\mathrm{ST}} := I_{\mathrm{ST}}$
  $\mathtt{Tx}_i^{\mathrm{d}} : M := I_m$
  $\mathtt{Rx}_i^{\mathrm{d}} : M :=$ empty

signatures
  output TX$(m : M)_i$
  input RX$(m : M)$
  internal PROCESS$()_i$

transitions
  output TX$(m)_i$
    pre:
      $\mathtt{now}_i^{\mathrm{a}} \leq t_1$
      $\neg\mathtt{Tx}_i$.empty() $\wedge f_{\mathrm{oc}}(\mathtt{ST}_i^{\mathrm{d}}) = i$
    eff:
      set $m := \mathtt{Tx}_i$
      set $\mathtt{Tx}_i$ empty
  input RX$(m : M)$
    eff:
      $\mathtt{Rx}_i^{\mathrm{d}} := m$

  internal PROCESS$()_i$
    pre:
      $t_2 \leq \mathtt{now}_i^{\mathrm{a}} \leq T$
    eff:
      $\mathtt{ST}_i^{\mathrm{d}} := f_{\mathrm{ST}}(\mathtt{ST}_i^{\mathrm{d}}, \mathtt{Rx}_i^{\mathrm{d}})$
      $\mathtt{Tx}_i^{\mathrm{d}} := f_{\mathrm{Tx}}(\mathtt{ST}_i^{\mathrm{d}}, m)$
      $\mathtt{now}_i^{\mathrm{a}} := 0$
      set $\mathtt{Rx}_i^{\mathrm{d}}$ empty

trajectories
stop when
  $\mathtt{now}_i^{\mathrm{a}} = t_1 \wedge f_{\mathrm{oc}}(\mathtt{ST}_i^{\mathrm{d}}) = i \wedge \neg\mathtt{Tx}_i^{\mathrm{d}}$.empty()
  $\mathtt{now}_i^{\mathrm{a}} = T$
evolve
  $d(\mathtt{now}_i^{\mathrm{a}}) := 1$
```

Figure 3.3: TIOA model of a generic node i.

Considering the timing behavior of the generic node as explained previously, the TIOA model in Fig. 3.3 already fulfills P1 to P6 in Section 3.1.1.

We further introduce the TIOA model SM in Fig. 3.4 of the shared-medium broadcast network in Fig. 3.1. We model the shared-medium broadcast network by a TIOA with the name SM

and the parameters n and M in Fig. 3.4. Here, n is a parameter of type *int* that identifies the number of nodes on the network, whereas M is a parameter that specifies the message format to be transmitted on the network: for each message m with type M, there is a field *m.length* that specifies the message length.

```
automaton SM(n : int, M : Type)

states                                   signature
 mess^d : M := empty                      input TX(m : M)_i
 coll^d : bool := false                   output RX(m : M)
 next^d : int := 0
 now^a : Real := 0

transitions                              output RX(m)
input TX(m)_i                             pre:
 eff:                                      (now^a = next^d) ∧ (mess^d not empty)
 if ((coll^d = false) ∧(mess^d is empty))  eff:
   mess^d := m                             set mess^d empty
   next^d:=m.length                        next^d := 0
 else
   coll^d := true
   next^d := 0
   set mess^d empty
 now^a := 0

trajectories
stop when                                evolve
 (now^a = next^d) ∧ (mess^d not empty)    d(now^a) = 1
```

Figure 3.4: TIOA for the shared-medium broadcast network.

The signature of the TIOA is given by the actions TX_i for $i = 1, \ldots, n$ and RX. TX_i is an input action that describes message passing from node i to the shared-medium broadcast network. Its parameter is the message m. The output action RX indicates that the transmission of a message is completed and the message is passed to all connected nodes.

The transition for TX_i has no pre-condition, since it belongs to an input action. Its effect depends on the state of SM. If no message is currently transmitted and no collision occurred previously, message transmission is started ($\mathtt{mess}^{\mathrm{d}} = m$) and the time for completion of the transmission is determined ($\mathtt{next}^{\mathrm{d}} := m.\mathrm{length}$). Otherwise, a collision is detected and all messages are discarded. In both cases, the analog variable $\mathtt{now}^{\mathrm{a}}$ is reset. The transition for the output action RX(m) occurs under the pre-condition that transmission of a message is completed and the parameter m is assigned the transmitted message ($\mathtt{now}^{\mathrm{a}} = \mathtt{next}^{\mathrm{d}}) \wedge (\mathtt{mess}^{\mathrm{d}}$ not empty) $\wedge (m = \mathtt{mess}^{\mathrm{d}})$. In that case the variables are updated such that currently no messages are transmitted.

Finally, the trajectories show that time evolves with a time derivative of 1 ($d(\mathtt{now}^{\mathrm{a}}) = 1$) and time stops when $(\texttt{now}^{\mathrm{a}} = \texttt{next}^{\mathrm{d}}) \wedge (\texttt{mess}^{\mathrm{d}}$ not empty$)$ in order to enforce $\textsc{sm2il}(m)$.

The behavior of SM is such that any node can transmit a message at any time. If the network is idle, the transmission is completed after the transmission delay of the message. Otherwise, a collision occurs, in which case SM shows further time evolution but does not allow any further actions.

Together, it holds that behavior of the overall system under consideration according to Fig. 3.1 is given by

$$SM||Node_1|| \cdots ||Node_n.$$

Respecting the definition of f_{oc}, it is readily observed that at most one node is allowed to transmit a message in each time slot. That is, there are no collisions on the shared-medium broadcast network. In addition, the definition of the shared action $\textsc{rx}$ in $Node_i$, $i = 1, \ldots, n$ and in SM ensures that each message that is transmitted by some node i before time t_1 is received by all nodes in the same time slot before time t_2. Then, data processing is done with the same received message by all nodes after time t_2. In order to also incorporate P7 in our model, we assume that data consistency should be achieved after processing. We introduce the function $f_{\mathrm{CON}} : A_{\mathrm{ST}} \times A_{\mathrm{ST}} \to \{0, 1\}$ such that for any state variables $ST, ST' \in A_{\mathrm{ST}}$, $f_{\mathrm{CON}}(ST, ST') = 1$ if and only if the data in ST and ST' is consistent. The consistency of the state information is application dependant. In our Node model, it depends on the equality of state variables in different nodes. Then, P7 holds if for any two nodes i and j, $f_{\mathrm{CON}}(\mathrm{ST}_i^{\mathrm{d}}, \mathrm{ST}_i^{\mathrm{d}}) = 1$ at time T in each time slot.

3.1.3 Potential Fault Scenarios

As is discussed in the previous section, the correct operation of the described system relies on the fact that the state variables of all nodes are consistent at the end of each time slot and that all transmitted messages are correctly received by all nodes. Although this result is formally correct by the design of the node model, it is obtained under the assumption of fault-free protocol operation. However, as is pointed out in the related literature [57, 5], the potential occurrence of software faults as well as hardware faults has to be taken into account. In principle such faults can originate in the software components of the system as well as in the underlying hardware and operating system layers if they are undetected in those layers [12]. We next assess possible faults to be expected during the system operation and classify these faults according to [1, 57]. We assume that faults occur *accidentally* during system operation and are neither caused in the *development phase* nor by an *incompetent user*. Hence, we consider the fault classes of *system boundaries* (internal or external), *persistence* (permanent or transient) and *dimension* (software or hardware) [57].

1. One possible fault is caused by the network link for example due to bit errors. Such external hardware fault is usually transient and has the effect that a transmission slot remains empty, that is, no message is received by any node. This effect is observed by

all network nodes since the allocation of transmission slots is known by the nodes using the function f_{oc} and in compliance with P5.

2. Another fault is potentially caused by wrong function evaluations of f_{ST} that lead to incorrect state update. Such internal software fault of a single node is usually permanent and is only observable in comparison to the state of other network nodes. The effect of such fault is the immediate or future inability of the faulty node to transmit messages in the correct time slot because of a wrong result of f_{oc} or f_{Tx}. Hence, it is possible that time slots are missed or collisions are caused.

3. It is further possible that software assets such as buffers or the operating system are faulty. Such internal software faults are usually transient but lead to permanent errors in the system state (similar to the case in 2). Hence, they cause the same effect.

4. Finally timing errors which cause transient synchronization problems can be encountered such that P1 is violated. Timing faults such as hardware or software timing jitters and delays, can lead to missed time slots or collisions. Moreover, such faults can cause the faulty node to transmit a message within a spare time slot. In either case, such fault can be detected by the connected healthy nodes within the network due to P5.

3.1.4 Dependability Design Requirements

The main objective of this chapter is the software fault-tolerance of distributed real-time systems as described in Section 3.1.2 with respect to faults as listed in Section 3.1.3. We next put forward several properties of the system under consideration that require particular attention in the dependability design in order to corroborate our design choices.

We first note that our system model has to be considered as a distributed real-time system with strong interaction between the system components. In particular, the state variables of all system components are supposed to be consistent and are potentially updated in each time slot, whereas the network bandwidth is limited. In addition, the correct protocol operation relies on message passing during fault-free operation, which complicates fault recovery [8]. Hence, we deduce the following requirements.

R1 Nodes should exchange messages for consistency checks of the state variables.

R2 Checkpoints should be taken frequently without need for additional message passing.

Our system model requires message exchange for its correct operation. In particular, the time slot ownership is determined by the system itself and there is no unrestricted network access for any node. This leads to the following requirements.

R3 Data exchange for dependability support should be piggybacked on application messages.

R4 Each node should/can only provide dependability data in its owned transmission slots.

Every node is aware of the ownership of any time slot. In particular, each node can identify if slots are used correctly or not if a message is sent in each transmission slot. Hence, we have the following requirement.

R5 A node should transmit a (potentially empty) message in each time slot it owns.

Under requirement R5, each node receives a message in each transmission slot. Hence, it is possible to perform a consistency check in each transmission slot.

R6 A consistency check should be performed at each node at the end of each transmission slot.

As described in Section 3.1.3, there can be faults that cause a faulty node to transmit within a spare time slot or cause a collision in a wrong transmission slot. Hence, a consistency check needs to be performed whenever a message is received on the shared medium and it is beneficial to prevent nodes that participate in a collision from sending further messages.

R7 A consistency check should be performed in each time slot where a message is received.

R8 Nodes that participate in a collision should avoid further transmissions untill the fault situation is solved.

3.2 Distributed Rollback Strategy with Synchronized Checkpointing

This section defines our dependability design depending on the facts and the node model given in Section 3.1. We first give our assumptions for the dependability design in Section 3.2.1. Then the basic operation of our strategy is defined in Section 3.2.2. Our design depends on an acceptance test mechanism for consistency checks. Section 3.2.3: defines the acceptance test realization. Then we give the generic model of our Dependability Plane in Section 3.2.4.

3.2.1 Assumptions

We first formulate realistic assumptions that should be satisfied, such that the described checkpointing and rollback strategy is successful.

It is not possible to detect system faults if the number of faulty nodes in the system exceeds a certain limit. As given in [58, 59] the total number of nodes should be more than twice of the number of faulty nodes. Hence the first assumption is:

A1 The relation between the number of nodes N in the system and the number of concurrent faulty nodes F is given as follows: $N \geq 2 \cdot F + 1$.

In order to begin the operation the system needs a baseline that is guaranteed to be fault free. Otherwise the system can not define a healthy checkpoint to roll-back in case of a system fault. Hence our second assumption is:

A2 There are no configuration faults at the beginning of the system operation such that the system is fault free at initialization.

3.2.2 Basic Operation

Respecting the stated requirements, we propose a novel distributed rollback strategy with synchronized coordinated checkpointing in order to achieve dependability of our distributed real-time system. The main features of our strategy are

F1 A *dependability plane* (DP) that is connected to the shared-medium broadcast network and to the software process on each node.

F2 A *dependability header* (DH) that is piggybacked on messages transmitted on the broadcast network.

F3 *Synchronized checkpointing* (SC) in order to store local copies of the state variables $\mathsf{ST}_i^{\mathrm{d}}$.

F4 An *acceptance test* (AT) at the end of each transmission slot and erroneously occupied spare slot.

F5 A *rollback message* (RM) that is transmitted in the first transmission slot of a node that is certain about a fault occurrence.

More precisely, the DP is realized as depicted in Fig. 3.5, whereby it is assumed that the DP constitutes a secure storage. At the beginning of each time slot, the valuation of the current state variables $\mathsf{ST}_i^{\mathrm{d}}$ is stored by the DP. Moreover, DP keeps track of the slot ownership as decided by f_{oc}. That is, our dependability design ensures frequent SC without any message exchange in line with R2.

It is required to check the consistency of the state variables $\mathsf{ST}_i^{\mathrm{d}}$ according to R1. This is achieved by DP that includes a copy of the state variables of the sender node (see Fig. 3.6) in compliance with R3, R4 and R5.

In principle, the described system operation ensures that a message is sent in each transmission slot in case of correct operation. That is, it is possible for each node to compare its state variables with the received DP. Accordingly, each node performs the acceptance test (AT) at the end of each transmission slot in view of R6. In addition, our DP checks each time slot in order to see whether it is occupied. In compliance with R7, the DP also triggers the AT in case of an unexpected message reception within a spare time slot. Furthermore, the DP disallows further transmissions of a node that participates in a collision according to R8.

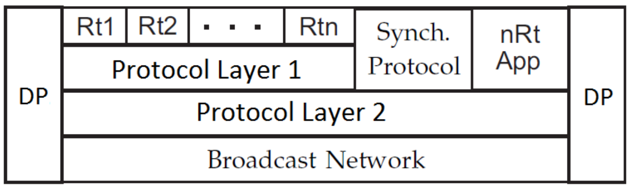


Figure 3.5: Dependability Plane Illustration

AT determines the local view of each node about the necessity of a rollback action and stores this information in a state variable $\mathtt{atDL}_i^d$ of DP. If a node with a rollback decision obtains a transmission slot, it sends a rollback message, where the DP in Fig. 3.6 contains $\mathtt{atDL}_i^d = false$. This message is received by all nodes such that all nodes roll back to a correct state that is stored in the DP. After the rollback operation, the system operation continues correctly from the rollback state. It has to be noted that the proposed strategy addresses the previously stated requirements R1 to R8. Since the AT and the resulting rollback action are the most critical components of our strategy, a detailed description of the AT based on the potential fault scenarios according to Section 3.1.3 is given in the sequel.

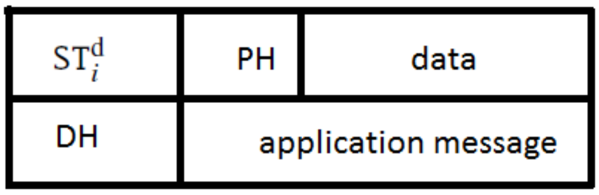


Figure 3.6: Frame encapsulation: DH (dependability header); PH (protocol header) and application data.

3.2.3 Acceptance Test Realization

From the local point of view of a generic node *i*, we identify the following observations that should lead to an unsuccessful AT and hence cause rollback.

O1 The content of the state variable ST_i^d of node *i* is not consistent with the variable received in DP of a correctly transmitted message of another node.

O2 No messages are received within a RT or nRT transmission slot.

O3 An unexpected message comes within a spare slot.

There is at least one possible scenario that can cause each of the above observations. In case O1, there can be two reasons for the variable mismatch. Either the receiver node or the transmitter node has corrupted state variable $\mathrm{ST}_i^{\mathrm{d}}$. In case O2, we identify three possible causes: a faulty receiver node can wrongly assume that a spare slot is a transmission slot, a faulty transmitter node can miss its transmission slot or a faulty node can use a transmission slot that is allocated to another node, whereas the resulting collision is also observed as an unoccupied transmission slot. In case O3, there are two possible fault scenarios: either the receiver node is faulty and considers a transmission slot as spare, or the transmitter node is faulty considering a spare slot as it's transmission slot.

According to the DP design, all of the above observations should cause an immediate rollback request to a correct time slot by the owner of the next transmission slot. In this context, two issues have to be taken into account. First, it has to be considered that, even if a fault is observed at a certain time slot, it is not known when the fault actually occurred. In order to evaluate this delay, we use the *worst-case inter-transmission delay* Δ_{I} as introduced in Section 3.1.2. Any faulty node will obtain network access within at most Δ_I time slots. Second, it is possible that the transmission of the rollback request is delayed. In the worst case, F nodes are prevented from transmitting a rollback request due to an observed collision. In that case, it is required to wait until the node $F + 1$ owns a transmission slot. That is, we introduce the *worst-case F-node transmission delay* Δ_{T} which is the maximum number of time slots until $F + 1$ different nodes own at least one transmission slot. Note that both Δ_I and Δ_T are application-dependent parameters that can be determined for each practical distributed real-time system realization. Together, the *worst-case fault detection delay* Δ_{F} evaluates to

$$\Delta_{\mathrm{F}} = \Delta_{\mathrm{I}} + \Delta_{\mathrm{T}}. \tag{3.1}$$

It is readily observed that, whenever a rollback request is transmitted by a node at a slot x, this implies that slot $x - \Delta_{\mathrm{F}}$ was correct in all nodes since it is either before or right at the correct previous transmission slot of the faulty node according to (3.1). Moreover, it can be deduced that the latest time for a fault-recovery is Δ_{F} time slots after a fault-occurrence. Denoting the duration of a time slot as $dSlot$, this leads to a worst-case recovery-time of $dSlot \times \Delta_{\mathrm{F}}$.

In view of the previous discussion, DP keeps a state variable history of Δ_{F} time slots and the AT checks for the situations defined in O1-O3 in every transmission slot and reception slot. If a node observes a faulty condition, it sets a rollback request flag and waits for its next transmission slot. Then, the node with the next transmission slot transmits the rollback request message. According to (3.1), the fault happened at most Δ_{I} time slots in the past and transmission is delayed at most Δ_{T} such that the at least the oldest item on the state variable history is correct. With the rollback request, each node rolls back to this oldest state. We note that a node that has raised a rollback request flag, does not repeatedly carry out the AT until a rollback occurs.

After the informal description of the dependability plane operation in this section, we formally model our dependability plane in the form of TIOA in Fig. 3.7.

TIOA $DP_i(\Delta_F : int, t_1 : int, t_2 : int, T : int, A_{ST} : Type, A_{DL} : Type, vDL : A_{DL}, A_{CS} : Type, M : Type)$

states

```
$\mathtt{now}_i^{a} : R := T$
$\mathtt{Hist}_i^{d} : Q[A_{ST}] := empty$
$\mathtt{atDL}_i^{d} : B := true$
$\mathtt{ocDL}_i^{d} : int := 0$
$\mathtt{recDL}_i^{d} : B := false$
$\mathtt{atRes}_i^{d} : B := true$
$\mathtt{stpTx}_i^{d} : B := false$
$\mathtt{vCS}_i^{d} : A_{CS} := empty$
```

signatures

```
output REQST($atRes: B$)$_i$
input UPDST($vST: A_{ST}$)$_i$
input RX($m: M$)
internal AT()$_i$
output RBACK($vST: A_{ST}$)$_i$
```

transitions

```
input UPDST($vST$)$_i$
 eff:
  if $\mathtt{Hist}_i^{d}$.Size() = $\Delta_F$
   $\mathtt{Hist}_i^{d}$.RemoveLast()
  $\mathtt{Hist}_i^{d}$.Push($vST$)
  $\mathtt{ocDL}_i^{d}$ =: $f_{oc}(vST.\mathrm{CS}, vDL)$

internal AT()$_i$
 pre:
  $\mathtt{atRes}_i^{d} = true \wedge \mathtt{now}_i^{a} = t_2$
  $\mathtt{ocDL}_i^{d} = true \vee \mathtt{recDL}_i^{d} = true$
 eff:
  if $f_{CS}$(Hist.Top().CS,
  $\mathtt{vCS}_i^{d}) \neq true$
   $\mathtt{atDL}_i^{d} := false$
  if $\mathtt{ocDL}_i^{d} \neq \mathtt{recDL}_i^{d}$
   $\mathtt{atDL}_i^{d} := false$
  if $(\mathtt{ocDL}_i^{d} = i) \wedge \neg\mathtt{recDL}_i^{d}$
   $\mathtt{stpTx}_i^{d} := true$
  $\mathtt{recDL}_i^{d} := false$

input RX($m$)
 eff:
  $\mathtt{vCS}_i^{d} := m.\mathrm{CS}$
  $\mathtt{atRes}_i^{d} := m.\mathrm{atRes}$
  $\mathtt{recDL}_i^{d} := true$

output RBACK($\mathtt{Hist}_i^{d}$.Last())$_i$
 pre:
  $\mathtt{atRes}_i^{d} = false$
  $\mathtt{now}_i^{a} = t_2$
 eff:
  $\mathtt{Hist}_i^{d}$.Clear()
  $\mathtt{atDL}_i^{d} := true$
  $\mathtt{ocDL}_i^{d} := 0$
  $\mathtt{recDL}_i^{d} := false$
  $\mathtt{stpTx}_i^{d} := false$
  $\mathtt{vCS}_i^{d} := empty$

output REQST($atRes$)$_i$
 pre:
  $\mathtt{now}_i^{a} = T$
 eff:
  $\mathtt{now}_i^{a} := 0$
```

trajectories

stop when

$(\mathtt{atRes}_i^{d} = true) \wedge (\mathtt{now}_i^{a} = t_2) \wedge (\mathtt{ocDL}_i^{d} = true \vee \mathtt{recDL}_i^{d} = true)$

$(\mathtt{atRes}_i^{d} = false) \wedge (\mathtt{now}_i^{a} = t_2)$

evolve

$d(\mathtt{now}_i^{a}) := 1$

Figure 3.7: Dependability Plane as TIOA.

The header parameters of DP_i are the worst-case fault detection delay Δ_F, the timing parameters t_1, t_2, T, the dependability plane input parameter vDL. The data types passed as automaton parameters are A_{ST}, A_{DL}, A_{CS} and M for history keeping variables, dependability plane state variables, message transmitting node state variables and message variables respectively.

Discrete transitions of DP_i are defined by the actions UPDST, AT, RX, RBACK and REQST. The input UPDST action is used to receive the local state information of the connected Node i automaton whereas the RX input action is used to receive the state information of the message transmitting node. DP_i carries an acceptance test which is triggered by the internal AT action transition. Here the automaton makes a concistency check between the local and received state informations. The output RBACK action is used to trigger a rollback request to the local Node i and finally the REQST action is used to request the local state information of Node i.

3.2.4 Dependability Plane Operation

This Section defines the operational steps of the Dependability Plane. The operation of the whole system (Node || DP || SM) is shown as a message sequence diagram in Fig. 3.8.

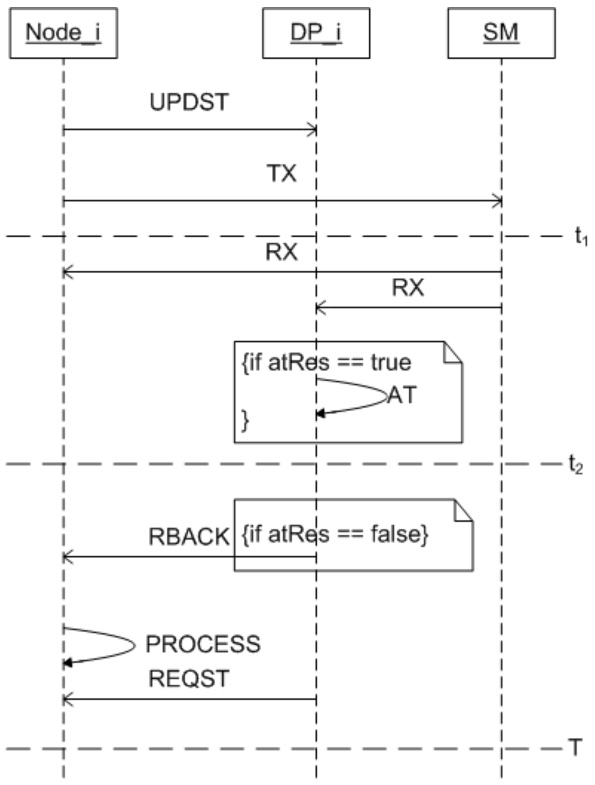


Figure 3.8: Message Sequence Diagram of the RT System Operation.

As seen in Fig. 3.8, the operation is started by the Node automaton. It transmits its current state information (via UPDST action) to DP automaton. Then it transmits a message (via TX action) on which the acceptance test result of the previous time slot as well as the current state information is piggybacked. Shared Medium (SM) automaton forwards the incomming message to each of the connected nodes via the RX action. The message information is forwarded both to the node and the DP automaton. DP behavior, after receiving the incoming message, varies depending on the acceptance test result (atRes) in the message. DP either triggers an acceptance test (via AT action) to see the current result of the network or triggers a rollback action via the RBACK action. At the end of each time slot, the node automaton calculates the state variables via the PROCESS action) while the DP automaton makes a request (via the REQST action) to the Node automaton for the next time slot.

The node automaton given in Fig. 3.3 is updated according to the dependability requirements. The updated node model is given in Fig. 3.9.

TIOA $Node_i(t_1 : int, t_2 : int, T : int, A_{\mathrm{ST}} : Type, I_{\mathrm{ST}} : A_{\mathrm{ST}}, M : Type, I_m : M)$

```
states
  $\mathtt{now}_i^{\mathrm{a}} : R := 0$
  $\mathtt{ST}_i^{\mathrm{d}} : A_{\mathrm{ST}} := I_{\mathrm{ST}}$
  $\mathtt{Tx}_i^{\mathrm{d}} : M := I_m$
  $\mathtt{Rx}_i^{\mathrm{d}} : M :=$ empty
  $\mathtt{updST}_i^{\mathrm{d}} : B := false$
  $\mathtt{stpTx}_i^{\mathrm{d}} : B := false$
  $\mathtt{atRes}_i^{\mathrm{d}} : B := true$

signatures
  output TX$(m : M)_i$
  input RX$(m : M)$
  internal PROCESS$()_i$
  input REQST$(tRes : B)_i$
  output UPDST$(ST : A_{\mathrm{ST}})_i$
  input RBACK$(rbST : A_{\mathrm{ST}})_i$

transitions
  input REQST$(tRes)$
    eff:
      $\mathtt{updST}_i^{\mathrm{d}} := true$
      $\mathtt{atRes}_i^{\mathrm{d}} := tRes$

  output UPDST$(\mathtt{ST}_i^{\mathrm{d}})$
    pre:
      $\mathtt{updST}_i^{\mathrm{d}} = true \wedge \mathtt{now}_i^{\mathrm{a}} \leq t_1$
    eff:
      $\mathtt{updST}_i^{\mathrm{d}} := false$

  output TX$(m)_i$
    pre:
      $\mathtt{now}_i^{\mathrm{a}} \leq t_1 \wedge f_{\mathrm{oc}}(\mathtt{ST}_i^{\mathrm{d}}) = i \wedge \mathtt{atRes}_i^{\mathrm{d}} = true$
    eff:
      set $m := \mathtt{Tx}_i$
      set $m.atRes = \mathtt{atRes}_i^{\mathrm{d}}$
      set $m.ST = \mathtt{ST}_i^{\mathrm{d}}$
      set $\mathtt{Tx}_i$ empty

  input RX$(m : M)$
    eff:
      $\mathtt{Rx}_i^{\mathrm{d}} := m$

  input RBACK$(rbST : A_{\mathrm{ST}})$
    eff:
      $\mathtt{ST}_i^{\mathrm{d}} := rbSt$
      $\mathtt{now}_i^{\mathrm{a}} := 0$

  internal PROCESS$()_i$
    pre:
      $t_2 \leq \mathtt{now}_i^{\mathrm{a}} \leq T$
    eff:
      $\mathtt{ST}_i^{\mathrm{d}} := f_{\mathrm{ST}}(\mathtt{ST}_i^{\mathrm{d}}, \mathtt{Rx}_i^{\mathrm{d}})$
      $\mathtt{Tx}_i^{\mathrm{d}} := f_{\mathrm{Tx}}(\mathtt{ST}_i^{\mathrm{d}}, m)$
      $\mathtt{now}_i^{\mathrm{a}} := 0$
      set $\mathtt{Rx}_i^{\mathrm{d}}$ empty

trajectories
  stop when
    $\mathtt{updST}_i^{\mathrm{d}} = true \wedge \mathtt{now}_i^{\mathrm{a}} = t_1$
    $\mathtt{now}_i^{\mathrm{a}} = t_1 \wedge f_{\mathrm{oc}}(\mathtt{ST}_i^{\mathrm{d}}) = i \wedge \mathtt{atRes}_i^{\mathrm{d}} = true$
    $\mathtt{now}_i^{\mathrm{a}} = T$

  evolve
    $d(\mathtt{now}_i^{\mathrm{a}}) := 1$
```

Figure 3.9: TIOA model of a generic node i extended by dependability actions.

In summary, this chapter describes our dependability design for broadcast communication of distributed real-time systems in the form of a dependability plane. To this end, a generic node model for distributed real-time systems with broadcast communication is defined. We further describe the dependability requirements of the systems under discussion and define our dependability plane based on a dependability header that is piggybacked on messages. The main idea of our design depends on a rollback mechanism that is initiated after the failure of an acceptance test hat is run in each time slot. Moreover we show that our design works in the presence of at most F faulty nodes assuming that the total number of nodes in the system is given by $N \geq 2 \cdot F + 1$. Finally, we are able to compute the maximum time delay Δ_F until a fault recovery is achieved in a distributed real-time system implementing our design.

CHAPTER 4

CASE STUDY: DEPENDABILITY DESIGN FOR A DISTRIBUTED INDUSTRIAL REAL-TIME PROTOCOL FAMILY

This chapter gives the dependability plane design of a distributed real-time industrial protocol family depending on the design methodology given in Section 3. The dependability plane design is given as a TIOA with formal statements proving the correct operation of the layer design.

4.1 Protocol Framework

The protocol family as proposed in [10] is designed as a purely distributed protocol on a shared-medium broadcast network with a distributed clock synchronization algorithm. It supports RT communication, whereby it allows for dynamically changing the bandwidth allocation to different network nodes. That is, this protocol family is particularly useful for control applications that incorporate information about their instantaneous communication requirements. Although the protocol formulation is general, its potential realization is targeted for shared-medium Ethernet in combination with the industrial standard IEEE 1588 [56] for precise clock synchronization. We next give an overview of this protocol family and then describe the relevant information about the protocol architecture for our dependability extension.

4.1.1 Overview

The protocol stack proposed in [10] is depicted in Figure 4.1. It is designed to operate on a broadcast network such as *shared-medium Ethernet* and comprises two protocol layers – an *Interface Layer* (IL) and a *Coordination Layer* (CL). Here, IL implements time-slotted medium access for both RT and nRT traffic to the broadcast network. The CL is responsible for (i) deciding whether the current time slot is allocated to RT or nRT traffic and (ii) determining which node is eligible to transmit a message in case of RT slots. That is, it is foreseen that multiple applications such as distributed RT control applications can be connected to the CL, whereas all messages from nRT applications (such as diagnostics or high-level messages) are directly handled by the IL. In contrast to existing protocols, the protocol family in [10] allows for dynamically changing the slot allocation during run-time based on application specific data. Unique allocation of nRT slots is provided by the IL in this framework. We denote each slot that is assigned to a unique node as a *transmission slot* and all remaining slots as *spare slots*. In order to establish a common time base for the time-slotted operation, the framework further includes the use of a synchronization protocol.

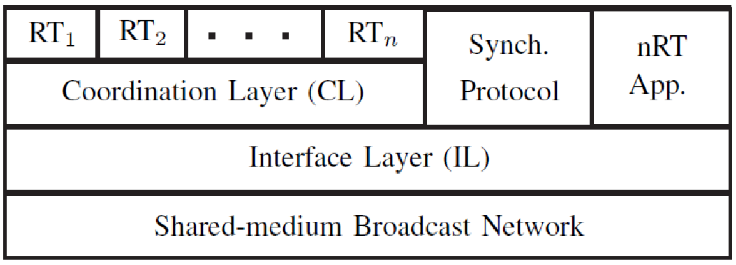


Figure 4.1: Software Architecture.

4.1.2 Shared-Medium Broadcast Network

The shared-medium broadcast network defined in Fig. 3.4 relies on the assumption that there is only one node transmitting a message at a certain time. Collision occurs in case of multiple message transmissions. The shared medium simply obtains an incoming message from the IL via the action *IL2SM*. Then, it forwards the message to the connected nodes by triggering the *SM2IL* action. In case of a second message arrival before forwarding a message, the medium simply raises a collision flag and both messages are dropped. The operation of the shared medium is compatible with conventional shared-medium Ethernet (IEEE 802.3).

4.1.3 Interface Layer

Considering that the protocol family is designed to be entirely distributed, each network node implements an identical IL. Hence, we consider the IL of a generic node i, denoted as IL_i, in the sequel. IL_i holds a set of state variables $\mathtt{vIL}_i^{\mathrm{d}}$ in order to decide about the ownership of nRT slots. The task of IL_i is to realize time-slotted access to the underlying broadcast network. In each time slot, the IL performs the following actions.

- Requesting RT message from CL (REQRT): The CL decides if the current slot is a RT slot that belongs to IL_i. So at the beginning of each time slot, IL_i asks CL_i for a RT message.

- RT data passing from CL (CL2ILRT): The CL decides if the current slot is a RT slot that belongs to IL_i. In the positive case, a RT message is stored in the RT transmit buffer of IL_i.

- nRT message passing from nRT applications (AP2ILNRT): As shown in Figure 4.1, IL_i has a direct connection with the nRT application running on top of the protocol stack. The nRT messages are passed directly to IL_i via the AP2ILNRT action that is triggered by the nRT application.
- Message passing to the broadcast network (IL2SM): The nRT slot ownership is decided by IL_i. If the current slot is a RT/nRT slot that belongs to IL_i, the message in the RT/nRT transmit buffer is transmitted to the shared medium.
- Message reception from the broadcast network (SM2IL): If a message m is received from the broadcast network in a RT slot, the received data is stored in a RT message buffer. Otherwise, the data is stored in a nRT message queue.
- RT message forwarding to the CL (IL2CLRT): If a RT message was received in the current slot, it is directly forwarded to the CL.
- nRT message forwarding to nRT applications IL2APNRT: nRT message transmission to the nRT application is controlled by the application itself. All the nRT messages that are buffered in IL_i are forwarded to the application via IL2APNRT input action that is triggered by the nRT application.
- Layer update (UPDATE): In order to get ready for the next time-slot, the state variable $\mathtt{vIL}_i^{\mathrm{d}}$ is updated at the end of each time slot.

Since the IL of each node performs the same computations and receives the same data from the connected CL, it is ensured that the IL state variables in all nodes are synchronized [10]. As a consequence, unless a fault occurs, a unique node is identified for transmission of both RT and nRT messages in each time slot. Moreover, considering that all messages are transmitted on a broadcast network, the IL of each node receives the same message in each time slot. We further note that the IL operation relies on a precise clock synchronization of all nodes that is performed by a synchronization protocol as depicted in Figure 4.1. In a practical implementation, the distributed synchronization protocol IEEE 1588 [56] can be used for this purpose as is exemplified in [60].

4.1.4 Coordination Layer

Analogous to the IL, the CL operation of each individual network node is identical. Hence, we describe the CL of a generic node i, denoted as CL_i. The task of CL_i is to forward RT messages from the application layer to the IL and vice versa, decide about the type of each time slot (RT or nRT) and uniquely determine the ownership of each RT slot. To this end, CL_i holds state variables $\mathtt{vCL}_i^{\mathrm{d}}$ that are updated in each time slot. The operation of CL_i is as follows.

- Message reception from RT applications (AP2CL): CL_i has an interface with the RT application running on the protocol stack. Available RT messages are received from the connected RT applications. Each message is placed in the transmit buffer of the respective application.

- Reception of RT message request from IL (REQRT): The slot type (RT or nRT) and slot ownership are decided based on the state variables $\mathsf{vCL}_i^{\mathsf{d}}$. CL_i internally computes the slot type and ownership information when the REQRT input action is triggered by IL_i

- RT message passing to IL (CL2ILRT): If the current slot is a RT slot and belongs to node i, an application message in the transmit buffer of CL_i is forwarded to IL_i.

- RT message reception from IL (IL2CLRT): RT messages that belongs to node i are received from IL_i. The internal state variable $\mathsf{vCL}_i^{\mathsf{d}}$ is updated according to the current state of CL_i and incoming RT message.

- RT message forwarding to the RT applications (CL2AP): The received RT message at the top of the receive buffer is forwarded to the RT application via the input action CL2AP triggered by the connected RT applications.

Since the CL of each node performs the same computations and receives the same data from the connected IL, it is ensured that the CL state variables are identical in all nodes [10]. As a consequence, unless a fault occurs, the CL always suggest a unique node for the message transmission in RT slots.

4.1.5 Protocol Operation

In order to clarify the protocol operation, we summarize the sequential actions that are taken by the different layers in each time slot as depicted in Figure 4.2 which is showing the action sequences within a typical time slot.

Considering Figure 4.2 if it is a RT time slot, the RT application transmits a message to CL via the AP2CL transition. IL_i asks for an available RT message to CL_i via REQRT. CL makes its internal computations to decide the type and the owner of the time-slot within this transition. When it decides that it is a RT time-slot reserved for node i, it informs IL_i by transmitting the RT message via the CL2ILRT transition. IL_i after taking the RT message, immediately forwards it to SM by IL2SM output action. SM, after taking the incoming message from node i, transmits the message to the connected nodes including node i via SM2IL transition. Then IL_i forwards the message to CL_i by the output IL2CLRT action transition. After CL_i takes the RT message, the RT application obtains the RT message via CL2AP transition. The operation within one time-slot ends with the internal UPDATE transition of IL_i,

Second, if it is a nRT time-slot, as shown in Figure 4.2 one of the connected nRT applications is willing to transmit a nRT message. At the beginning of the time-slot, the nRT application

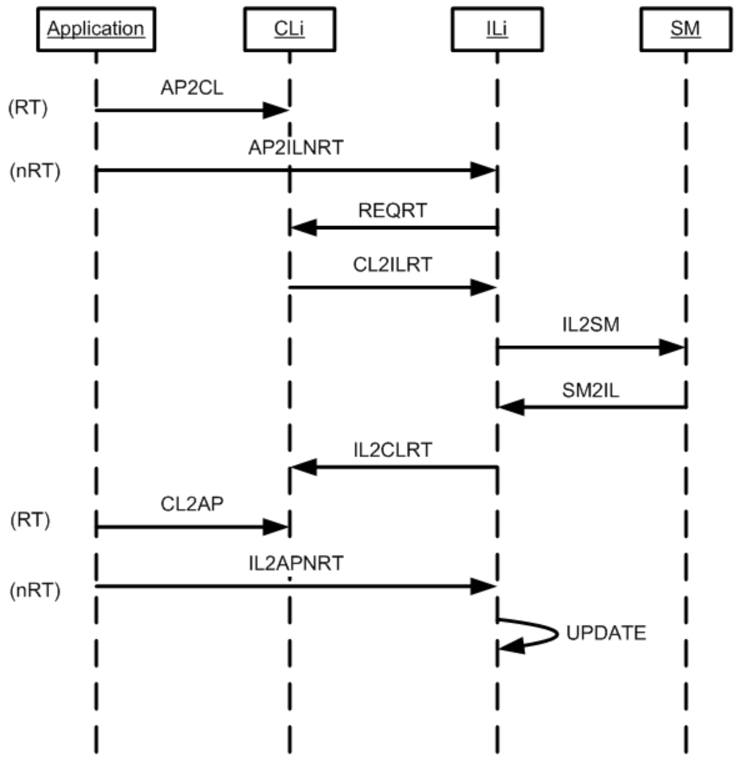


Figure 4.2: Operational Sequence within a typical time slot

transmits a message to IL via the AP2ILNRT transition. IL_i asks for an available RT message to CL_i via REQRT. CL makes its internal computations to decide the type and the owner of the time-slot within this transition. When it decides that it is not a RT time-slot, it informs IL_i by transmitting an empty RT message via the CL2ILRT transition. IL_i after taking the empty message makes its internal computations to determine whether it is a nRT time-slot reserved for node i. If the decision is positive, it immediately forwards the message at the top of the *nRT* message queue to SM by IL2SM output action. SM, after taking the incoming message from node i, transmits the message to the connected nodes including node i via SM2IL transition. After IL_i takes the nRT message, the nRT application gets the nRT message via IL2APNRT transition. The operation within one time-slot ends with the internal UPDATE transition of IL_i.

We finally note that the same actions are performed in each time slot, whereby always at most one node has the right to transmit and all other nodes listen to and process the messages transmitted on the broadcast network. As an important feature of the proposed protocol, all RT messages are received by the CL of each node such that the CL data are always consistent among all nodes. Moreover, the operation of the IL of each node maintains consistent information about the ownership of nRT slots.

Before starting the dependability design, we first check whether the protocol family satisfies the necessary properties defined in Section 3.1.1. First of all regarding P1, the distributed nodes in the defined framework are synchronized by the software synchronization protocol

IEEE1588. The protocol family implements a time-slotted shared medium access such that each node in the network transmits a message within a time slot. Hence it satisfies P2 and P3. The protocol framework implements a distributed resource allocation such that the medium access schedule is determined locally at each node in a synchronized fashion. In order to avoid message collisions the medium access schedule is consistent among distributed nodes. Hence there is a consistent state information (P7) in the distributed nodes and by this information each node knows the message transmission instants (P5). Moreover scheduling mechanism can be defined in order each node to have a regular medium access fulfilling the required property defined in P4. Finally, regarding P6 the local state information at each node is updated at the end of each time-slot. In summary, our distributed protocol exhibits all the properties defined in Section 3.1.1.

4.2 TIOA Modeling of D^{3}RIP Framework

The D^{3}RIP protocol family is designed to support the dependable operation of networking protocols for distributed industrial applications having both real-time and non-real-time communication. In particular, D^{3}RIP formalizes the dependability extension for the distributed real-time protocol framework defined in the previous section.

The protocol stack given in Figure 4.3 is designed to operate on a broadcast network such as *shared-medium Ethernet*. It defines three protocol layers – an *Interface Layer* (IL), a *Coordination Layer* (CL) and a *Dependability Plane* (DP). Here, IL and CL realize the actual protocol functionality as previously described in [10].

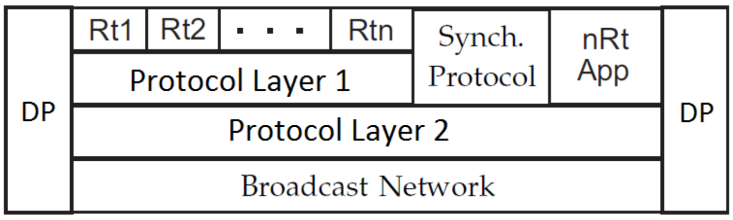


Figure 4.3: Framework Architecture Including DP

The major difference of D^{3}RIP to the existing protocol family is the dependability plane that is targeted to achieve software fault-tolerance. In this section, we formalize our suggested DP in the TIOA framework. In particular, we extend the existing TIOA models of SM, IL and CL in order to realize the distributed synchronized checkpointing/rollback recovery strategy described in Section 3.2. In addition, we define the new DP in the form of a TIOA.

4.2.1 Framework Layers

Considering that our framework is designed to be entirely distributed, each network node implements identical layers. Hence, we consider the IL, CL and DP of a generic node i, denoted as IL_i, CL_i and DP_i respectively.

4.2.1.1 Shared-Medium Broadcast Network (SM)

The SM TIOA model is given in Figure 4.4. SM is defined such that messages are dropped in case of collision (action IL2SM$_i$). Moreover, since SM is connected to both IL and DP, the action SM2ILDL is shared with IL_i and DP_i for $i = 1, \ldots, n$.

TIOA $SM(N : int, M : type)$

```
states
 mess^d : M := empty
 next^d : int := 0
 now^a : Real := 0

signature
 input IL2SM(m : M)_i
 output SM2ILDL(m : M)

transitions
input IL2SM(m)_i
 eff:
 if (mess^d is empty)
   mess^d := m
   next^d := m.length
 else
   next^d := 0
   set mess^d empty
 now^a := 0

output SM2ILDL(m)
 pre:
  (now^a = next^d) ∧ (mess^d not empty)
 eff:
  set mess^d empty
  next^d := 0

trajectories
stop when
 (now^a = next^d) ∧ (mess^d not empty)
evolve
 d(now^a) = 1
```

Figure 4.4: TIOA for the shared-medium broadcast channel.

4.2.1.2 Interface Layer (IL)

The TIOA model of the IL is given in Figure 4.5. In order to realize the interface between IL and DP, new states and actions are introduced in comparison to the IL definition in [10]. The layer updates are implemented according to the design rules defined in Section 3.2.4. The flags $\mathtt{sendvIL}_i^{\mathrm{d}}$, $\mathtt{stpTx}_i^{\mathrm{d}}$ and $\mathtt{at}_i^{\mathrm{d}}$ indicate the necessity of passing the state variables $\mathtt{vIL}_i^{\mathrm{d}}$ to DP_i, the necessity of not transmitting any more messages and the success of the latest acceptance test, respectively. The output action UPDVIL$_i$ is defined to carry $\mathtt{vIL}_i^{\mathrm{d}}$ to DP_i at the fixed time $t_{\mathrm{IL}1}$ in each time slot for synchronized checkpointing. The input action SENDRES$_i$ provides the results of the acceptance tests from DP_i and updates $\mathtt{stpTx}_i^{\mathrm{d}}$ and $\mathtt{at}_i^{\mathrm{d}}$ accordingly. The input action RBACK is triggered by DP_i and receives correct IL and CL rollback state information $ilHT$ and $clHT$. Then, IL_i rolls back to the received state by updating $\mathtt{vIL}_i^{\mathrm{d}}$, $\mathtt{TxnRT}_i^{\mathrm{d}}$ and $\mathtt{RxnRT}_i^{\mathrm{d}}$. The input action SM2ILDL receives the messages coming from SM at each transmission slot.

4.2.1.3 Coordination Layer (CL)

The TIOA model of CL is given in Figure 4.6. In comparison to [10], the new state $\mathtt{sendvCL}_i^{\mathrm{d}}$ is introduced. It is true if the state variable $\mathtt{vCL}_i^{\mathrm{d}}$ has to be passed to DP. This is performed by the output action UPDVCL at time $t_{\mathrm{CL}0}$ after REQRT. Similar to IL, the input action RBACK rolls back to the correct state $clHT$ supplied by DP_i. The states $\mathtt{vCL}_i^{\mathrm{d}}$, $\mathtt{Tx}_i^{\mathrm{d}}$ and $\mathtt{Rx}_i^{\mathrm{d}}$ are updated accordingly.

4.2.2 Dependability Plane (DP)

As described in Section 3.2, the layer operation proposes a novel distributed rollback strategy with synchronized coordinated checkpointing in order to achieve dependability of our framework. We now present our dependability plane implementation given in Figure 4.7.

Considering the general formulation of IL and CL in the form of a protocol family, our DP is general in the sense that its realization can be adapted to the respective member of the protocol family. To this end, we introduce a parameter vDL and functions f_{oc}, f_{CL} and f_{IL}.

DP design fits the structure of the general dependability plane design in Fig. 3.7. UPDVIL and UPDVCL actions implements the UPDST action of the general DP automaton. AT action of the general model is implemented by the ATEST action. Finally, SM2DL, RBACK and REQST actions of the general DP automaton are implemented by the SM2ILDL, RBACK and SENDRES actions respectively.

Parameters: The header parameters of DP_i are the worst-case fault detection delay Δ_{F}, the timing parameters $t_{\mathrm{DL}0}$, $t_{\mathrm{DL}1}$, the data types A_{IL} and A_{CL} for IL and CL state variables, respectively, and the dependability plane input parameters vDL of data type A_{DL}. vDL contains data related to the respective member of the protocol family and is used for consistency checks in the AT.

States: The time evolution is captured by the analog state $\mathtt{now}_i^{\mathrm{a}}$. The remaining variables are all discrete. The queues $\mathtt{ILHist}_i^{\mathrm{d}}$ and $\mathtt{CLHist}_i^{\mathrm{d}}$ keep a bounded history of the IL and CL state respectively, and the integer $\mathtt{stNo}_i^{\mathrm{d}}$ manages the internal action transitions. $\mathtt{stNo}_i^{\mathrm{d}}$ is set to 1 if an acceptance test is required, is set to 2 if information needs to be passed to the IL and is set to 3 if rollback is required. Otherwise, $\mathtt{stNo}_i^{\mathrm{d}} = -1$. The boolean $\mathtt{atDL}_i^{\mathrm{d}}$ keeps the acceptance test result, the boolean $\mathtt{ocDL}_i^{\mathrm{d}}$ indicates if the current slot is a transmission slot

TIOA $IL_i(dSlot\colon int, t_{\mathrm{IL}0}\colon int, t_{\mathrm{IL}1}\colon int, t_{\mathrm{IL}2}\colon int, t_{\mathrm{IL}3}\colon int, M : Type, Q : Type, A_{IL} : Type, H_{IL} : Type, InitIL\colon A_{\mathrm{IL}})$

states

```
$\mathtt{now}^{a}_{i}\colon R := dSlot$; $\mathtt{TxRT}^{d}_{i}\colon M :=$ empty
$\mathtt{TxnRT}^{d}_{i}\colon Q :=$ empty; $\mathtt{RxRT}^{d}_{i}\colon M :=$ empty
$\mathtt{RxnRT}^{d}_{i}\colon Q :=$ empty; $\mathtt{RTIL}^{d}_{i}\colon B :=$ false
$\mathtt{myIL}^{d}_{i}\colon B :=$ false; $\mathtt{reqIL}^{d}_{i}\colon B :=$ false
$\mathtt{vIL}^{d}_{i}\colon A_{IL} :=$ InitIL; $\mathtt{sendvIL}^{d}_{i}\colon B :=$ false
$\mathtt{stpTx}^{d}_{i}\colon B :=$ false; $\mathtt{at}^{d}_{i}\colon B :=$ true
```

signature

```
output IL2SM$(m\colon M)_i$; output REQRT$()_i$
input CL2ILRT$(b_{my}\colon B, b_{RT}\colon B, m\colon M)_i$
input AP2ILNRT$(m\colon M)_i$; input IL2APNRT$(q\colon Q)_i$
output IL2CLRT$(m\colon M)_i$; internal UPDATE$()_i$
output UPDVIL$(vIL\colon A_{IL}, TxnRT\colon Q,$
$RxnRT\colon Q, myIL\colon B)_i$
input SENDRES$(atRes\colon B, stpTx\colon B)$
input RBACK$(ilHT\colon H_{\mathrm{IL}}, clHT\colon H_{\mathrm{CL}})$
input SM2ILDL$(m\colon M)$
```

transitions

```
internal UPDATE$()_i$
 pre:
  $\mathtt{now}^{a}_{i} = dSlot$
 eff:
  $\mathtt{vIL}^{d}_{i} = f_{\mathrm{upd}}(\mathtt{vIL}^{d}_{i}, \mathtt{RTIL}^{d}_{i})$
  $\mathtt{now}^{a}_{i} := 0$
  $\mathtt{reqIL}^{d}_{i} :=$ true
  $\mathtt{sendvIL}^{d}_{i} :=$ true

output UPDVIL$(\mathtt{vIL}^{d}_{i}, \mathtt{TxnRT}^{d}_{i}, \mathtt{RxnRT}^{d}_{i}, \mathtt{myIL}^{d}_{i})_i$
 pre:
  $\mathtt{sendvIL}^{d}_{i} = \mathrm{true} \wedge$
  $\mathtt{now}^{a}_{i} = t_{\mathrm{IL}1}$
 eff:
  $\mathtt{sendvIL}^{d}_{i} :=$ false

input SENDRES$(atRes, stpTx)_i$
 eff:
  $\mathtt{at}^{d}_{i} := \mathtt{atRes}$
  $\mathtt{stpTx}^{d}_{i} := stpTx$

input RBACK$(ilHT, clHT)_i$
 eff:
  $\mathtt{vIL}^{d}_{i} := ilHT.\mathrm{vIL}$
  $\mathtt{TxnRT}^{d}_{i} := ilHT.\mathrm{TxnRT}$
  $\mathtt{RxnRT}^{d}_{i} := ilHT.\mathrm{RxnRT}$
  $\mathtt{now}^{a}_{i} := 0$
  $\mathtt{reqIL}^{d}_{i} :=$ true
  $\mathtt{sendvIL}^{d}_{i} :=$ true
  $\mathtt{at}^{d}_{i} :=$ true

output REQRT$()_i$
 pre:
  $\mathtt{reqIL}^{d}_{i} = \mathrm{true} \wedge$
  $\mathtt{now}^{a}_{i} = t_{\mathrm{IL}0}$
 eff:
  $\mathtt{reqIL}^{d}_{i} =$ false

input CL2ILRT$(b_{\mathrm{my}}, b_{\mathrm{RT}}, m)_i$
 eff:
  $\mathtt{RTIL}^{d}_{i} = b_{\mathrm{my}}$
  $\mathtt{myIL}^{d}_{i} = f_{\mathrm{my}}(\mathtt{vIL}^{d}_{i}, \mathtt{RTIL}^{d}_{i}, b_{\mathrm{RT}}, i)$
  $\mathtt{TxRT}^{d}_{i} = m$

input IL2APNRT$(\mathrm{RxnRT}^{d}_{i})_i$
 eff:
  set $\mathrm{RxnRT}^{d}_{i}$ empty

input AP2ILNRT$(m)_i$
 eff:
  $\mathtt{TxnRT}^{d}_{i}$.Push$(m)$

output IL2SM$(m)_i$
 pre:
  $\mathtt{now}^{a}_{i} = t_{\mathrm{IL}2} \wedge \mathtt{myIL}^{d}_{i} \wedge \neg\mathtt{stpTx}^{d}_{i} \wedge (\neg(\mathtt{TxRT}^{d}_{i}$ empty$) \wedge$
  $\mathtt{RTIL}^{d}_{i}) \vee (\neg\mathtt{RTIL}^{d}_{i} \wedge \neg(\mathtt{TxnRT}^{d}_{i}$.Top() empty$))$
 eff:
  if $\mathtt{RTIL}^{d}_{i}$
      set $m = \mathtt{TxRT}^{d}_{i}$
      set $\mathrm{TxRT}^{d}_{i}$ empty
  else
      set $m = \mathtt{TxnRT}^{d}_{i}$.Top()
      $\mathtt{TxnRT}^{d}_{i}$.Pop()
      $m.\mathrm{vCL} = \mathtt{TxRT}^{d}_{i}.\mathrm{vCL}$
  set $m.\mathrm{atRes} = \mathtt{at}^{d}_{i}$
  set $m.\mathrm{vIL} = \mathtt{vIL}^{d}_{i}$
  $\mathtt{myIL}^{d}_{i} =$ false

input SM2ILDL$(m)$
 eff:
  if $\mathtt{RTIL}^{d}_{i}$
      $\mathrm{RxRT}^{d}_{i} = m$
  else
      $\mathrm{RxnRT}^{d}_{i}$.Push$(m)$

output IL2CLRT$(m)_i$
 pre:
  $\mathtt{now}^{a}_{i} = t_{\mathrm{IL}3} \wedge$
  $\neg(\mathrm{RxRT}^{d}_{i}$ empty$)$
 eff:
  set $m = \mathrm{RxRT}^{d}_{i}$
  set $\mathrm{RxRT}^{d}_{i}$ empty
```

Trajectories $\mathcal{T}$

```
stop when
 $(\mathtt{now}^{a}_{i} = dSlot) \vee ((\mathtt{now}^{a}_{i} = t_{\mathrm{IL}0}) \wedge (\mathtt{reqIL}^{d}_{i} = \mathrm{true})) \vee (\mathtt{sendvIL}^{d}_{i} \wedge (\mathtt{now}^{a}_{i} = t_{\mathrm{IL}1}))$
 $((\mathtt{now}^{a}_{i} = t_{\mathrm{IL}2}) \wedge \mathtt{myIL}^{d}_{i} \wedge \neg\mathtt{stpTx}^{d}_{i} \wedge (\neg(\mathtt{TxRT}^{d}_{i}$ empty$) \wedge \mathtt{RTIL}^{d}_{i}) \vee (\neg\mathtt{RTIL}^{d}_{i} \wedge \neg(\mathtt{TxnRT}^{d}_{i}$.Top empty$)))$
 $\mathtt{now}^{a}_{i} = t_{\mathrm{IL}3} \wedge \neg(\mathrm{RxRT}^{d}_{i}$ empty$)$
evolve
 $d(\mathtt{now}^{a}_{i}) = 1$
```

Figure 4.5: IL model as TIOA

```
TIOA $CL_i(del_i\colon int, t_{\mathrm{CL0}}\colon int, M\colon Type, Q\colon Type, V\colon Type, A_{\mathrm{CL}}\colon Type, H_{\mathrm{CL}}\colon Type, InitCL\colon A_{\mathrm{CL}})$
states
  $\mathtt{send}_i^{\mathrm{a}}\colon R := del_i$
  $\mathtt{Tx}_i^{\mathrm{d}}\colon V :=$ empty
  $\mathtt{Rx}_i^{\mathrm{d}}\colon Q :=$ empty
  $\mathtt{RTCL}_i^{\mathrm{d}}\colon B :=$ false
  $\mathtt{myCL}_i^{\mathrm{d}}\colon B :=$ false
  $\mathtt{ch}_i^{\mathrm{d}}\colon int := 0$
  $\mathtt{reqCL}_i^{\mathrm{d}}\colon B\colon =$ false
  $\mathtt{vCL}_i^{\mathrm{d}}\colon A_{\mathrm{CL}}\colon = InitCL$
  $\mathtt{sendvCL}_i^{\mathrm{d}}\colon B\colon =$ false

signatures
  input AP2CL$(m\colon M, ch\colon int)_i$
  input IL2CLRT$(m\colon M)_i$
  input REQRT$()_i$
  output CL2ILRT$(RTCL\colon B,$
    $myCL\colon B, m\colon M)_i$
  input CL2AP$(q\colon Q)_i$
  output UPDVCL$(vCL\colon A_{CL}, Tx\colon V, Rx\colon Q,$
    $RTCL\colon B)_i$
  input RBACK$(ilHT\colon H_{\mathrm{IL}}, clHT\colon H_{\mathrm{CL}})$

transitions
  input AP2CL$(m, ch)_i$
    eff:
      $\mathtt{Tx}_i^{\mathrm{d}}[ch] := m$

  input IL2CLRT$(m)_i$
    eff:
      $\mathtt{Rx}_i^{\mathrm{d}}$.Push$(m)$
      $\mathtt{vCL}_i^{\mathrm{d}} := g_{\mathrm{upd}}(\mathtt{vCL}_i^{\mathrm{d}}, m)$

  input REQRT$()_i$
    eff:
      $\mathtt{RTCL}_i^{\mathrm{d}} = g_{\mathrm{RT}}(\mathtt{vCL}_i^{\mathrm{d}}, t)$
      $(\mathtt{myCL}_i^{\mathrm{d}}, \mathtt{ch}_i^{\mathrm{d}}) := g_{my}(\mathtt{vCL}_i^{\mathrm{d}}, i)$
      $\mathtt{send}_i^{\mathrm{a}} := 0$
      $\mathtt{reqCL}_i^{\mathrm{d}} :=$ true
      $\mathtt{sendVCL}_i^{\mathrm{d}} :=$ true

  output UPDVCL$(\mathtt{vCL}_i^{\mathrm{d}}, \mathtt{Tx}_i^{\mathrm{d}}, \mathtt{Rx}_i^{\mathrm{d}}, \mathtt{RTCL}_i^{\mathrm{d}})_i$
    pre:
      $\mathtt{sendVCL}_i^{\mathrm{d}} =$ true$\wedge$
      $\mathtt{send}_i^{\mathrm{a}} = t_{\mathrm{CL0}}$
    eff:
      $\mathtt{sendVCL}_i^{\mathrm{d}} :=$ false

  input RBACK$(ilHT, clHT)_i$
    eff:
      $\mathtt{vCL}_i^{\mathrm{d}} := clHT.vCL$
      $\mathtt{Tx}_i^{\mathrm{d}} := clHT.Tx$
      $\mathtt{Rx}_i^{\mathrm{d}} := clHT.Rx$

  output CL2ILRT$(\mathtt{RTCL}_i^{\mathrm{d}}, \mathtt{myCL}_i^{\mathrm{d}}, m)_i$
    pre:
      $\mathtt{reqCL}_i^{\mathrm{d}} \wedge (\mathtt{send}_i^{\mathrm{a}} = del_i)$
    eff:
      if $\mathtt{myCL}_i^{\mathrm{d}}$
        $m := \mathtt{Tx}_i^{\mathrm{d}}[\mathtt{ch}_i^{\mathrm{d}}]$
        set $\mathtt{Tx}_i^{\mathrm{d}}[\mathtt{ch}_i^{\mathrm{d}}]$ empty
      else
        set $m$ empty
      $m$.vCL := $\mathtt{vCL}_i^{\mathrm{d}}$
      $\mathtt{reqCL}_i^{\mathrm{d}} :=$ false

  input CL2AP$(\mathtt{Rx}_i^{\mathrm{d}})_i$
    eff:
      set $\mathtt{Rx}_i^{\mathrm{d}}$ empty

trajectories
  stop when
    $(\mathtt{sendVCL}_i^{\mathrm{d}} = true) \wedge (\mathtt{send}_i^{\mathrm{a}} = t_{\mathrm{CL0}})$
    $(\mathtt{reqCL}_i^{\mathrm{d}} = true) \wedge (\mathtt{send}_i^{\mathrm{a}} = del_i)$
  evolve
    $d(\mathtt{send}_i^{\mathrm{a}}) := 1$
```

Figure 4.6: Coordination Layer as TIOA.

and the boolean $\mathtt{recDL}_i^{\mathrm{d}}$ captures if a message was received in the current slot. The boolean $\mathtt{mySlot}_i^{\mathrm{d}}$ indicates if the current slot is owned by node i or not and the boolean $\mathtt{stpTx}_i^{\mathrm{d}}$ decides if further transmissions should be prohibited. Finally, $\mathtt{rvCL}_i^{\mathrm{d}}$ and $\mathtt{rvIL}_i^{\mathrm{d}}$ store the received CL and IL state variables respectively.

Signatures and Transitions: The input transition UPDVCL$_i$ is triggered at the beginning of the time slot by CL_i. It carries the state variables vCL and message buffers Tx, Rx of CL_i in addition to the boolean variable $RTCL$ to indicate the type of the current time-slot (RT or nRT). The checkpoint vCL, Tx, Rx is stored in $\mathtt{CLHist}_i^{\mathrm{d}}$, keeping the queue length bounded by Δ_{F}. Moreover, the slot is considered as occupied ($\mathtt{ocDL}_i^{\mathrm{d}} = true$) if it is an RT slot. The time variable $\mathtt{now}_i^{\mathrm{a}}$ is reset to 0 in order to initiate the cyclic operation of DP_i.

The input transition UPDVIL is triggered by IL_i. It carries the state variables $\mathtt{vIL}_i^{\mathrm{d}}$ in addition to the slot ownership information $myIL$ and the message buffers $TxnRT$ and $RxnRT$. DP_i

```
TIOA $DP_i(\Delta_F : int, t_{DL0} : int, t_{DL1} : int, A_{IL} : Type,\ A_{CL} : Type, A_{DL} : Type, vDL : A_{DL})$
states
 $now^a_i : R := t_{DL1}$
 $ILHist^d_i : Q[A_{IL}] := empty$
 $CLHist^d_i : Q[A_{CL}] := empty$
 $stNo^d_i : I := -1$
 $atDL^d_i : B := true$
 $ocDL^d_i : B := false$
 $recDL^d_i : B := false$
 $mySlot^d_i : B := false$
 $stpTx^d_i : B := false$
 $rvCL^d_i : A_{CL} := empty$
 $rvIL^d_i : A_{IL} := empty$
signatures
 input UPDVCL$(vCL\colon A_{CL}, Tx\colon V, Rx\colon Q, RTCL\colon B)_i$
 input UPDVIL$(vIL\colon A_{IL}, myIL : B, TxnRT\colon Q, RxnRT\colon Q)_i$
 input SM2ILDL$(m\colon M)$
 internal ATEST$()_i$
 output RBACK$(ilH\colon A_{IL}, clH\colon A_{CL})_i$
 output SENDRES$(atRes\colon B, stpTx\colon B)_i$
transitions
 input UPDVCL$(vCL, Tx, Rx, RTCL)_i$
  eff:
   $now^a_i := 0$
   if $CLHist^d_i$.Size() = $\Delta_F$
    $CLHist^d_i$.RemoveLast()
   $CLHist^d_i$.Push$(vCL, Tx, Rx)$
   if $RTCL = true$
    $ocDL^d_i := true$
   else
    $ocDL^d_i := false$
 input UPDVIL$(vIL, myIL, TxnRT, RxnRT)_i$
  eff:
   if $ILHist^d_i$.Size() = $\Delta_F$
    $ILHist^d_i$.RemoveLast()
   $ILHist^d_i$.Push$(vIL, TxnRT, RxnRT)$
   if $ocDL^d_i = false$
    $ocDL^d_i =: f_{oc}(vIL, vDL)$
   if $ocDL^d_i = true$
    $stNo^d_i := 1$
   $mySlot^d_i := myIL$
 input SM2ILDL$(m)$
  eff:
   $rvCL^d_i := m.vCL$
   $rvIL^d_i := m.vIL$
   $recDL^d_i := true$
   if $m$.atRes $= false$
    $stNo^d_i := 3$
   else
    $stNo^d_i := 1$
 internal ATEST$()_i$
  pre:
   $stNo^d_i = 1 \land atDL^d_i = true \land$
   $now^a_i = t_{DL0}$
  eff:
   if $f_{CL}$(CLHist.Top().$vCL$, rvCL) $\neq true$
    $atDL^d_i := false$
   if $f_{IL}$(ILHist.Top().$vIL$, rvIL) $\neq true$
    $atDL^d_i := false$
   if $ocDL^d_i \neq recDL^d_i$
    $atDL^d_i := false$
   if $mySlot^d_i \land \neg recDL^d_i$
    $stpTx^d_i := true$
   $stNo^d_i := 2$
   $recDL^d_i := false$
 output SENDRES$(atDL^d_i, stpTx^d_i)_i$
  pre:
   $stNo^d_i = 2 \land$
   $now^a_i = t_{DL1}$
  eff:
   $stNo^d_i := -1$
 output RBACK$(ILHist^d_i$.Last(), $CLHist^d_i$.Last()$)_i$
  pre:
   $stNo^d_i = 3 \land$
   $now^a_i = t_{DL0}$
  eff:
   $ILHist^d_i$.Clear()
   $CLHist^d_i$.Clear()
   $stNo^d_i := -1$
   $atDL^d_i := true$
   $ocDL^d_i := false$
   $recDL^d_i := false$
   $stpTx^d_i := false$
trajectories
 stop when
  $(stNo^d_i = 1) \land atDL^d_i = true \land (now^a_i = t_{DL0})$
  $(stNo^d_i = 2) \land (now^d_i = t_{DL1})$
  $(stNo^d_i = 3) \land (now^a_i = t_{DL0})$
 evolve
  $d(now^a_i) := 1$
```

Figure 4.7: Dependability Plane as TIOA.

stores the checkpoint vIL, $TxnRT$, $RxnRT$ in $\texttt{ILHist}_i^{\mathrm{d}}$, keeping the queue length bounded by Δ_{F}. If the slot is a nRT slot ($\texttt{ocDL}_i^{\mathrm{d}} = false$), the slot occupancy is decided based on the information from the dependability plane input parameter vDL. Here, vDL has to contain information about the nRT slot ownership of other nodes since vIL only keeps information about the slot ownership of node i [10]. The slot occupation information is decided via the function $\texttt{f}_{\mathrm{oc}}(vIL, vDL)$, whose realization depends on the specific protocol implementation. Example realizations are given in Chapter 6. If the slot is occupied, the AT is requested ($\texttt{stNo}_i^{\mathrm{d}} = 1$). The $\texttt{mySlot}_i^{\mathrm{d}}$ variable is set according to the incoming information from IL_i.

DP_i receives messages coming from SM by the input transition SM2ILDL. DP_i stores the received state variables $m.vCL$ and $m.vIL$ in the respective states and checks if a rollback is requested (m.atRes $= false$). In the positive case, $\texttt{stNo}_i^{\mathrm{d}}$ is set to 3 to trigger the output RBACK transition. Otherwise $\texttt{stNo}_i^{\mathrm{d}}$ is set to 1 to trigger the internal ATEST transition.

The internal transition ATEST evaluates the consistency between the local system state and the observed system state. The acceptance test fails if (i) the local state variables and the received state variables (CL and IL) are inconsistent, (ii) the expected slot occupancy is different from the observed message reception ($\texttt{ocDL}_i^{\mathrm{d}} \neq \texttt{recDL}_i^{\mathrm{d}}$) or (iii) a collision happens in the transmission slot of node i ($\texttt{mySlot}_i^{\mathrm{d}} \wedge \neg\texttt{recDL}_i^{\mathrm{d}}$). In the latter case, also $\texttt{stpTx}_i^{\mathrm{d}} = true$ is set in order to stop further transmissions. In addition, we note that the functions f_{CL} and f_{IL} are used to perform the consistency check of CL and IL variables, respectively. These functions are defined such that only in case of non-faulty operation in any time slot and for any two different nodes $i \neq j$

$$f_{\mathrm{CL}}(\texttt{vCL}_i^{\mathrm{d}}, \texttt{vCL}_j^{\mathrm{d}}) = true \tag{4.1}$$

$$f_{\mathrm{IL}}(\texttt{vIL}_i^{\mathrm{d}}, \texttt{vIL}_j^{\mathrm{d}}) = true \tag{4.2}$$

The realization of these functions depends on the respective member of our protocol family. An example realization is given in Chapter 6.

The output transition SENDRES is triggered after each ATEST transition. DP_i sends the AT result and the decision about further transitions $\texttt{stpTx}_i^{\mathrm{d}}$ to IL_i.

The output transition RBACK is triggered if $\texttt{stNo}_i^{\mathrm{d}}$ has the value 3 (rollback is requested by a transmitter node). In that case, DP_i provides the correct rollback state information of IL_i and CL_i that is stored as the last entry of $\texttt{ILHist}_i^{\mathrm{d}}$ and $\texttt{CLHist}_i^{\mathrm{d}}$, respectively. Furthermore, all other values are reset.

4.2.3 D^{3}RIP Operation

We next summarize the overall operation of our dependable protocol family in each time slot. To this end, we use the message sequence diagram in Figure 4.8.

If it is a RT time-slot, the application transmits a RT message via the AP2CL transition to CL_i.

On the other hand, if it is a nRT time-slot the application transmits a nRT message to IL_i via the AP2ILNRT transition.

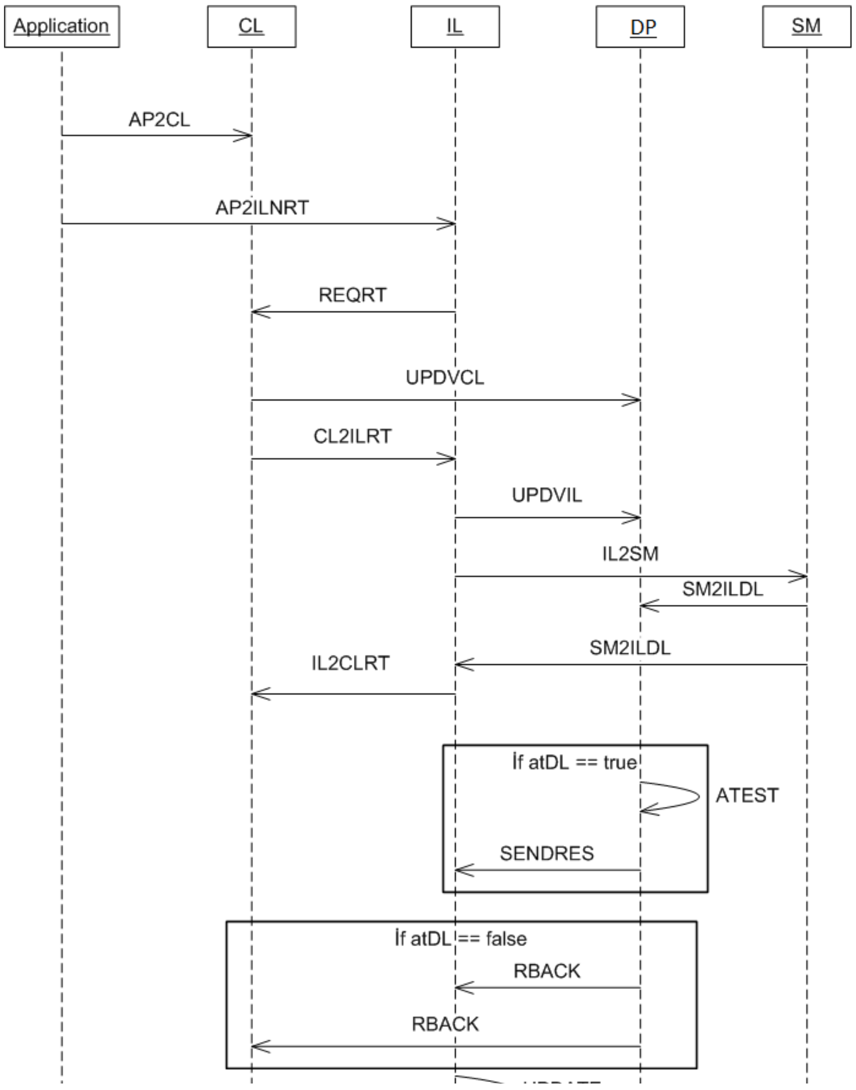


Figure 4.8: D^{3}RIP Operation

Then, at the very beginning of the time slot, IL_i sends a RT message request to CL_i via the REQRT. After REQRT, CL_i transmits its state information to DP_i via UPDVCL. In addition, CL_i finishes its computations and passes the slot type to IL_i via CL2ILRT. In a RT transmission slot that is owned by node i, CL_i transmits a RT message to IL_i. Otherwise, an empty message only with the state variable information $\mathsf{vCL}_i^{\mathrm{d}}$ is provided via CL2ILRT.

After executing CL2ILRT, IL_i sends its state information to DP_i with UPDVIL. In RT slots that are owned by node i, IL_i sends the received RT message via IL2SM. Otherwise, IL_i decides if it owns the nRT slot and transmits a nRT message to SM in the positive case via IL2SM.

A message received by SM is immediately forwarded to IL and DP of the connected nodes by SM2ILDL. DP_i has two choices depending on the information stored in the dependability header of the message. If there is a rollback request within the message, it triggers RBACK to execute a rollback in IL_i and CL_i. If there is no rollback request, DP_i executes the internal ATEST and sends its result to IL_i via SENDRES.

When there is no rollback request, there are two possible operations for IL_i. In case of a RT slot, IL_i transmits the incoming message to CL_i via IL2CLRT. Otherwise, IL_i stores the incoming message until it is polled by a nRT application. The time slot is completed and a new time slot is started with the clock reset in IL_i that is triggered by the internal UPDATE transition of IL_i.

The incoming message is transmitted to the application via the CL2AP transition in case of a RT time-slot. In case of a nRT time-slot the application gets the nRT message via the IL2APNRT transition. The designed protocol family with the Dependability Plane is implemented in [61].

It is important to configure the IL, CL and DP automaton timings in order to avoid race conditions. The timing configuration is dependent on the hardware platform that the protocol family is running on. Hence with the correct timing configuration of the automatons the protocol operation given in Fig. 4.8 does not face with race conditions.

4.2.4 Formal Results

We finally state several desirable properties of our dependability design with their formal proofs. To this end, we investigate the composition

$$DP\_CL\_IL\_SM = SM||(||_{i\in\mathcal{I}}IL_i)||(||_{i\in\mathcal{I}}CL_i)||(||_{i\in\mathcal{I}}DP_i) \tag{4.3}$$

of DP, CL, IL and the shared-medium broadcast network, as illustrated in Figure 4.9.

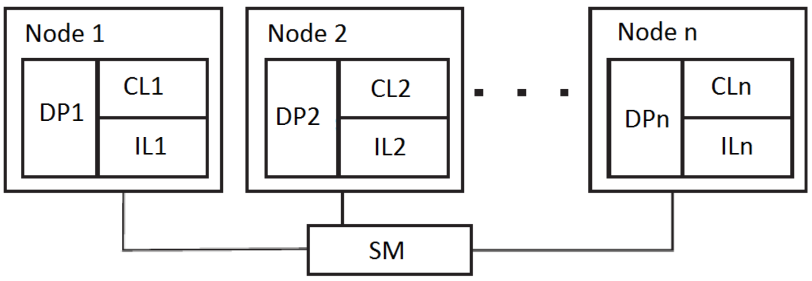


Figure 4.9: Communication via DP,CL,IL and SM

4.2.4.1 Progressiveness

The DP given in Fig. 4.7, as well as the overall protocol family is progressive. Hence it does not have infinite number of transitions within a finite time interval. Depending on the composition relation defined in Section 2.3, the overall protocol family is progressive. That is, it is ensured that a finite number of transitions are taken in each finite time interval.

Lemma 1 *DP as given in Fig. 4.7 is progressive.*

Proof 1 *The locally controlled actions are* ATEST, SENDRES *and* RBACK. ATEST *is only executed at time t_{DL0} if $\mathtt{stNo}_i = 1$ and $\mathtt{stNo}_i := 2$ is set with each occurrence. Hence, an infinite number of* ATEST *occurrences requires infinite time. An analogous argument holds for* SENDRES *at time t_{DL1} and $\mathtt{stNo}_i = 2$ (reset to $\mathtt{stNo}_i := -1$) and* RBACK *at time t_{DL0} and $\mathtt{stNo}_i = 3$ (reset to $\mathtt{stNo}_i := -1$).*

It holds that the overall protocol family is progressive. That is, it is ensured that a finite number of transitions are taken in each finite time interval.

Proposition 1 *$DP\_CL\_IL\_SM$ as depicted in Figure 4.9 is progressive.*

Proof 2 *It is shown in [10] that $CL\_IL\_SM = SM||(||_{i\in\mathcal{I}}IL_i)||(||_{i\in\mathcal{I}}CL_i)$ is progressive. Since DP_i is progressive for all $i = 1,\dots,n$ according to Lemma 1, Theorem 1 in Section 2.3 directly implies that DP_CL_IL_SM is progressive.*

4.2.4.2 Synchronized Checkpointing

We next address the consistency of the synchronous checkpoints that are taken by DP. That is, the contents of the DP variables are consistent in all time slots as long as no fault occurs.

Proposition 2 *Assume that a successful rollback action is performed latest Δ_F time slots after a fault occurrence. Then, it holds for any $k \in \mathbb{N}_0$ at $k \cdot dSlot + t_{IL0} + t_{CL0} + t_{DL0}$ that $f_{CL}(\mathtt{CLHist}^{\mathrm{d}}_i.Last(), \mathtt{CLHist}^{\mathrm{d}}_j.Last()) = true$, $\mathtt{stNo}^{\mathrm{d}}_i = \mathtt{stNo}^{\mathrm{d}}_j$ and $f_{IL}(\mathtt{ILHist}^{\mathrm{d}}_i.Last(), \mathtt{ILHist}^{\mathrm{d}}_j.Last()) = true$ for any $k \in \mathbb{N}_0$ and $i, j \in \mathcal{I}$.*

Lemma 2 *It holds that* UPDVIL$_i$, *$i \in \mathcal{I}$ occurs synchronized at times $k \cdot dSlot + t_{IL1}$ and* UPDVCL$_i$, *$i \in \mathcal{I}$ occurs synchronized at times $k \cdot dSlot + t_{IL0} + t_{CL0}$ for $k \in \mathbb{N}_0$.*

Proof 3 *The locally controlled output action* UPDVIL *of IL_i $i \in I$ occurs exactly once in each time slot t_{IL1} time units after the associated internal action* UPDATE *is triggered since the precondition $\mathtt{sendvIL}^d_i$ is disabled after one occurrence. Since the local clocks $\mathtt{now}^{\mathrm{a}}_i$ and $\mathtt{now}^{\mathrm{a}}_j$ of any nodes $i, j \in \mathcal{I}$ evolve synchronously,* UPDVIL$_i$ *also occurs synchronously for all $i \in \mathcal{I}$ in each time slot.*

Similarly the locally controlled output action UPDVCL *of CL_i, $i \in \mathcal{I}$, can only occur once in each time slot t_{CL0} time units after the associated input action* REQRT, *whereas* REQRT *is triggered at the times $k \cdot dSlot + t_{IL0}$. Considering the synchronicity of the local clocks $\mathtt{send}_i^{\mathrm{a}}$ and $\mathtt{send}_j^{\mathrm{a}}$ for $i, j \in \mathcal{I}$,* UPDVCL$_i$ *occurs synchronously at $k \cdot dSlot + t_{IL0} + t_{CL0}$ in all nodes.*

Lemma 3 *It holds for any $k \in \mathbb{N}_0$ at time $k \cdot dSlot + t_{IL1}$ that $f_{CL}(\mathtt{CLHist}_i^{\mathrm{d}}.Top().vCL,$ $\mathtt{CLHist}_j^{\mathrm{d}}.Top().vCL) = true$ and $f_{IL}(\mathtt{ILHist}_i^{\mathrm{d}}.Top().vIL, \mathtt{ILHist}_j^{\mathrm{d}}.Top().vIL) = true$ for any non-faulty nodes $i, j \in \mathcal{I}$.*

Now it is possible to prove Proposition 2:

Proof 4 *We prove the assertion by induction. Initially it holds for all $i \in \mathcal{I}$ that $\mathtt{CLHist}_i^{\mathrm{d}}.Last() =$ empty, $\mathtt{ILHist}_i^{\mathrm{d}}.Last() =$ empty and $\mathtt{stNo}_i^{\mathrm{d}} = -1$. Now, assume that for some $k \in \mathbb{N}_0$ and time $k \cdot dSlot + t_{IL0} + t_{CL0} + t_{DL0}$, $f_{CL}(\mathtt{CLHist}_i^{\mathrm{d}}.Last().\mathtt{vCL}, \mathtt{CLHist}_j^{\mathrm{d}}.Last().\mathtt{vCL}) = true$, $f_{IL}(\mathtt{ILHist}_i^{\mathrm{d}}.Last().\mathtt{vIL}, \mathtt{ILHist}_j^{\mathrm{d}}.Last().\mathtt{vIL}) = true$ and $\mathtt{stNo}_i^{\mathrm{d}} = \mathtt{stNo}_j^{\mathrm{d}}$ for all $i, j \in \mathcal{I}$. Then next update for $\mathtt{CLHist}$ and $\mathtt{ILHist}$ happens with the occurrence of* UPDVCL$_i$ *(at time $(k+1) \cdot dSlot + t_{IL0} + t_{CL0}$) and* UPDVIL$_i$ *(at time $(k+1) \cdot dSlot + t_{IL1}$), respectively which are synchronized for all $i \in \mathcal{I}$ according to Lemma 3.*

There are two possible cases. In the first case, the number of entries of $\mathtt{CLHist}_i^{d}$ and $\mathtt{ILHist}_i^{d}$ is smaller than $\Delta_F - 1$. Then, according to the definition of UPDVCL *and* UPDVIL *in DP_i, it holds that $\mathtt{CLHist}_i^{\mathrm{d}}.Last()$ is empty and $\mathtt{ILHist}_i^{\mathrm{d}}.Last()$ is empty after the update. Hence, $f_{CL}(\mathtt{CLHist}_i^{\mathrm{d}}.Last().\mathtt{vCL}, \mathtt{CLHist}_j^{\mathrm{d}}.Last().\mathtt{vCL}) = true$ and $f_{IL}(\mathtt{ILHist}_i^{\mathrm{d}}.Last().\mathtt{vIL},$ $\mathtt{ILHist}_j^{\mathrm{d}}.Last().\mathtt{vIL}) = true$ also at time $(k+1) \cdot dSlot + t_{IL0} + t_{CL0} + t_{DL0}$. In the second case, the number of entries of $\mathtt{CLHist}_i^{d}$ and $\mathtt{ILHist}_i^{d}$ is at least $\Delta_F - 1$. That is, after the update with* UPDVCL *and* UPDVIL, *the entry at position $\Delta_F - 1$ moves to $\mathtt{CLHist}_i^{\mathrm{d}}.Last()$ and $\mathtt{ILHist}_i^{\mathrm{d}}.Last()$, respectively. Considering that this entry was stored Δ_F time slots in the past, it must be correct in all nodes by assumption (otherwise a successful rollback should have occurred). Again, this implies that $f_{CL}(\mathtt{CLHist}_i^{\mathrm{d}}.Last().\mathtt{vCL}, \mathtt{CLHist}_j^{\mathrm{d}}.Last().\mathtt{vCL}) = true$ and $f_{IL}(\mathtt{ILHist}_i^{\mathrm{td}}.Last().\mathtt{vIL},$
$\mathtt{ILHist}_j^{\mathrm{d}}.Last().\mathtt{vIL}) = true$ also at time $(k+1) \cdot dSlot + t_{IL0} + t_{CL0} + t_{DL0}$. This concludes the induction.*

4.2.4.3 Rollback

We finally investigate the properties of our rollback recovery strategy. It holds that an acceptance test is performed in every transmission slot and in every spare slot where a message is wrongly transmitted.

Proposition 3 *Consider $DP\_CL\_IL\_SM$ as defined in Equation* (4.3). *It is guaranteed that an acceptance test is performed in each transmission slot and in each slot where a message is transmitted.*

Proof 5 *DP_i triggers the internal* ATEST *action at exactly t_{DL0} time units after the* UPDVCL *input action.* ATEST *action trigger is controlled via the $\mathtt{stNO}_i^d$ variable which is set to* 1*, within* UPDVIL *action in case of a transmission slot and within* SM2ILDL *input action in case of a message reception from SM. This mechanism guarantees that an acceptance test is performed in each transmission slot and in each slot where a message is received.*

We finally point out that indeed every fault as specified in O1 to O3 (in Section 3.2.3) is tolerated with a successful rollback operation carried out within Δ_F time slots as is required by F5 in Section 3.2.2.

Theorem 2 *Consider DP_CL_IL_SM as defined in* (4.3)*. It is guaranteed that if a fault occurs at time slot k, the correct protocol operation is resumed at time slot $k + \Delta_F$ latest.*

Lemma 4 *DP design guarantees that, a rollback occurs at most Δ_F time slots after the fault occurrence.*

Proof 6 *Lemma is proved by induction. Assume that at a certain time instant a fault is occurred. Regarding the dependability design, at least DP_i of one node i recognizes the fault. Node i waits for its turn to transmit the rollback request which is at most Δ_I slot times. Then in the worst case scenario there exists a collision and Δ_T spare slots, before the third node transmits the rollback request, because of stopping 2 collided nodes. So there exists a total of $\Delta_F = \Delta_I + \Delta_T$ number of time slots until a fault is fixed by a* RBACK *request from a healty node.*

Lemma 5 *DP design guarantees that; if a rollback occurs latest Δ_F time slots after any fault occurrence then, rollback state is correct.*

Proof 7 *Assuming that the current time slot is x, as described in Lemma 4, the rollback request in the network can be delayed for at most $\Delta_F - 1$ time slots. Hence the state at time slot $x - \Delta_F$ is guaranteed to be a healthy state.*

Now the proof of Theorem 2 can be done depending on Lemma 4 and 5.

Proof 8 *A rollback request may occur within Δ_F time slots after the fault occurs. So rolling back for Δ_F time slots guarantees that the network will resume its correct operation.*

In this section we have given a protocol family, which fulfills the properties defined in Section 3.1.1. We have designed the dependability plane of the protocol family as an application example of our design. Finally we have concluded the section with the formal proofs showing the correctness of our dependability design defined in Chapter 3.

CHAPTER 5

MODELING DISTRIBUTED TIOA SYSTEMS IN UPPAAL

In this chapter, the task of generating networks of TA models from a composition of TIOA models is addressed. We first define our motivation in Section 5.1. Section 5.2 discusses several restrictions on TIOA models that have to be fulfilled in order to enable conversion to UPPAAL. Our conversion procedure the TUConvert Tool that implements it is defined in Section 5.3 . After the algorithm definition, in Section 5.4 we describe the Timed Automata semantics that is used in UPPAAL. Section 5.5 formally shows the correctness of our conversion scheme. We emphasize that our algorithm is suitable for distributed real-time systems, since it respects all synchronization properties of systems that are modeled by multiple synchronized TIOA.

5.1 TIOA Modeling in UPPAAL

Distributed real-time computing systems are designed to satisfy certain timing constraints in addition to the application level behavioral constraints. Such systems require the analysis and verification of the system properties before implementation.

To this end, timed input output automata (TIOA), as introduced in Section 2.3, forms a viable mathematical framework for modeling and analyzing real-time systems. Furthermore this framework is suitable for distributed systems as the timed behavior of a system can be represented by the *composition* of multiple TIOA representing its individual components. It is possible to check practical conditions for individual TIOA models such as *progressiveness* to ensure continued system operation. It is a nice property of the framework that progressiveness is preserved after composition.

Besides modeling, verification of a system design is an important branch of system design. Formal approaches are frequently employed to check whether a system design logically fulfills its timing and behavioral specifications [24, 25]. Specifically model-checking is a powerful technique that can provide complete proofs of specifications fulfillment. Model checking is usually carried out by comparing the behavioral model of a system with a temporal logic formula defining the system level requirements. Short verification time and reduced complexity with the possibility of partial function checking are the advantages of model-checking in the formal verification process [26, 27]. Model-checking requires to model the system behavior as a concurrent state-graph. UPPAAL [7] is a well-known automatic verification software tool which can be used to generate behavioral models of the distributed real-time models. It has been used to verify the operation of different real-time systems such as a robot scrub nurse [28], POSIX operating systems [29] and timed multitask systems [30]. UPPAAL uses timed automata extended with data variables as the description language. It also has a simulator that is used to see the behavior of the system.

It can be observed that, on the one hand, the TIOA framework provides convenient models for distributed real-time systems and one can manually construct formal proofs for certain system properties. On the other hand, UPPAAL is a software tool for formal verification and employs behavioral models of the system components which again should be constructed manually. The motivation for this part of our study is to achieve a complete workflow where the TIOA models for system components are automatically converted to UPPAAL templates and then formally verified by UPPAAL.

5.2 Assumptions for TIOA Models

Before presenting our conversion algorithm, we discuss several conditions, that should be fulfilled by a TIOA $\mathcal{A}$ in order to be converted to an UPPAAL model.

First, it has to be respected that UPPAAL only allows specifying a unique initial evaluation of the template variables.

Assumption 1 *The initial state set Q_0 of $\mathcal{A}$ is a singleton, that is $Q_0 = \{q_0\}$ with the unique initial state q_0.*

Second, with the same reasoning, it must be possible to represent the data types in $\mathcal{A}$ in UPPAAL.

Assumption 2 *The data types used in $\mathcal{A}$ can be represented in UPPAAL (a similar assumption is made in [62]).*

Third, other than the Statistical Model Checker, TA in UPPAAL are designed for the representation of time as a continuous variable with time derivative 1 or 0, whereas the general definition of TIOA allows using differential equations in order to describe the evolution of trajectories. Hence, we formulate the following assumption.

Assumption 3 *The trajectory evolution of each analog (clock) variable c^a of $\mathcal{A}$ is given by $d(c^a) = 1$ or $d(c^a) = 0$ (a similar assumption is made in [62]). Moreover, there are no arithmetic operations on clock variables except for "resets" $c^a := b$ in the effect part of $\mathcal{A}$, whereby b is smaller than the current valuation of c^a.*

UPPAAL allows resetting clock variables to a constant value such as 0. With Assumption 3 we avoid forward jumps in time in the TA operation.

Action preconditions and trajectory stop conditions are defined as predicate on the TIOA state. We assume that these predicates are given in a standardized form.

Assumption 4 *The precondition for the transition of each action* A *of* $\mathcal{A}$ *is given in DNF:*

$$f^{\mathrm{A}}(val(X)) = (f^{\mathrm{A}1}(val(X)) \vee \cdots \vee f^{\mathrm{A}l_{\mathrm{A}}}(val(X))) \wedge u^{\mathrm{A}}(Y_{\mathrm{A}})$$

with the clauses $f^{\mathrm{A}i}(val(X))$ *depending on the variable valuation* $val(X)$ *and the clause* $u^{\mathrm{A}}(Y_{\mathrm{A}})$ *depending on the optional parameters* Y_{A}. *The stop condition of* $\mathcal{A}$ *is given in DNF*

$$w(val(X)) = (w^1(val(X^d)) \wedge (x_1 = v^1(val(X^d)))) \vee \cdots \vee (w^m(val(X^d)) \wedge (x_m = v^m(val(X^d)))),$$

whereby $x_i \in X^a$ *for* $i = 1, \ldots, m$. *That is, literals with analog variables are defined as* $x_i = v^i(val(X^d))$, *meaning that the analog variable* $x_i \in X^a$ *reaches the value* $v^i(val(X^d))$.

We briefly discuss the implications of the different assumptions. Assumption 1 is a realistic assumption for practical systems that generally start from a unique initial state. Assumption 2 is not a severe restriction considering that a large number of relevant data types can be defined in the C++ programming language. Assumption 3 does not allow modeling systems with dynamic behavior different from the mere evolution of time. Although this is a restriction of generality, we show in Section 5.6 that practical systems such as communication protocols can be modeled under this assumption.[1] Finally, Assumption 4 allows for a convenient conversion of predicates to UPPAAL.

The TIOA models that are used to generate the UPPAAL models in Fig. 2.5 fulfill the above listed assumptions. First of all both automata have a unique initial state (Assumption 1). All the data types used in the TIOA models are well-defined in UPPAAL (Assumption 2). The analog variable $\texttt{now}^{\mathrm{a}}$ used in the TIOA models has a rate of change of 1 and there is no arithmetic operation on $\texttt{now}^{\mathrm{a}}$ (Assumption 3). Finally, input action preconditions of the corresponding TIOA models are defined in DNF with $w_1(val(X^{\mathrm{d}})) = (\texttt{maxout}^{\mathrm{d}} > 0)$ and $v_1(val(X^{\mathrm{d}})) = \texttt{next}^{\mathrm{d}}$ where the analog literal is given by $x_1 = \texttt{now}^{\mathrm{a}}$ (Assumption 4).

Based on the previous assumptions, it is now possible to state our conversion algorithm.

5.3 *TUConvert* Algorithm

The *TUConvert* algorithm takes a TIOA $\mathcal{A}$ as defined in Section 2.3 as input and produces the corresponding TA $\mathcal{U}$ in UPPAAL as the output. In the following, we provide a step-wise description of *TUConvert*. To this end, we discuss all features of a generic TIOA $\mathcal{A}$ and give

[1] Also note that ODEs could be included in the assumption in order to define hybrid automata for simulation in UPPAAL. However, we focus on the TIOA models that are suitable for model-checking.

an equivalent TA representation $\mathcal{U}$. We further relate our description to the pseudo-code of Algorithm 1.

TIOA Parameters

The TIOA definition contains a list of parameters P_F, that represent fixed values of a certain type and parameters P_T that denote a type. We directly pass parameters representing a fixed value of a certain type as parameters $F = P_F$ of the UPPAAL model in line 1 of Algorithm 1. On the other hand the parameters P_T are defined as global structures in UPPAAL in lines 2 to 5 of Algorithm 1.

TIOA Variables

Since the TIOA state variables X are local, we identify them with the *local variables* M of $\mathcal{U}$ as type(x) $v_x = x_0$;. The data type of each variable in M is simply chosen as the data type of the corresponding TIOA variable in X. Note that a representation of this data type in UPPAAL always exists because of Assumption 2. Also depending on Assumption 3, each analog variable of $\mathcal{A}$ is represented as a clock variable in $\mathcal{U}$ as `clock` $k_x = x_0$. In Algorithm 1, the local variables are defined between line 6 to 12. Note that *type*(x) denotes the data type of the respective TIOA variable x to be represented in UPPAAL.

Discrete Transitions of TIOA

In our UPPAAL template model, we define the location *l_idle* in line 25 of Algorithm 1. The discrete transitions in $\mathcal{A}$ are represented by self-loop edges originating from *l_idle* in $\mathcal{U}$. The basic structure is shown in Fig. 5.1.

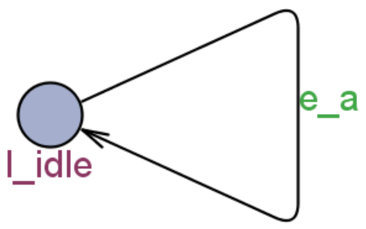


Figure 5.1: Transition representation in UPPAAL

That is, we assume that $\mathcal{U}$ stays in *l_idle* until the edge e_a is fired, which is identified with the actual execution of the corresponding transition t_a in $\mathcal{A}$. Based on this basic representation the main task is now constructing the edge definitions and constructing the fields of the edge e_a depending on the type of t_a.

For transitions t_a with an input action $a \in I$ we define a single edge e_a in line 32 of Algorithm 1. Here, we write for example $l\_idle \xrightarrow{e_a} l\_idle$ to characterize the selfloop edge e_a.

On the other hand, for the transitions t_a with an output or internal action $a \in O \cup H$ the number of defined edges depends on the number l_a of clauses in the DNF f^a of the corresponding transition precondition as defined in Assumption 4. For each clause f^{ai}, an edge e_{ai} is defined in line 37 of Algorithm 1.

Guard: The guard field of e_a corresponds to the preconditions of the corresponding transition t_a. Since there are no preconditions for transitions with input actions, no guard is defined for all t_a such that $a \in I$. In case $a \in O \cup H$ with the precondition $f^a(val(X)) = f^{a1}(val(X)) \vee \cdots \vee f^{al_a}(val(X))$, the guard condition e_{ai}.guard $= f^{ai}(val(M))$ is added to e_{a_i} in line 38 of Algorithm 1. Hereby, we use the fact that X in $\mathcal{A}$ is identified with the local variables M of $\mathcal{U}$.

Sync: The synchronization field of e_a determines which output channels are triggered or which input channels are waited for by e_a in order to perform synchronized actions. In the TIOA framework, it is required to synchronize transitions for external actions with common names among different TIOA. By definition, such transitions are triggered by the TIOA, whose transition belongs to an output action.

In the TA, we propose to introduce a broadcast channel $c_a \in C$ in $\mathcal{U}$ for each external action $a \in I \cup O$. This makes it possible to synchronize actions with the same name in different TIOA when converting transitions. The broadcast channels are introduced in Algorithm 1 in line 19 to 22. Regarding the synchronization, we use e_a.sync $= c_a?$ for actions $a \in I$. For actions $a \in H \cup O$, we use e_{ai}.sync $= c_a!$ for all $i = 1, \ldots, l_a$. The related statements In Algorithm 1 can be found in line 33 and 41.

Update: When a transition t_a in a TIOA $\mathcal{A}(P)$ is taken, the new valuation of the TIOA variables X is determined by the part "eff" of the TIOA representation.

If t_a belongs to an internal or output action $a \in H \cup O$, the new valuation of X only depends on the current valuation of X. This computation can be directly converted to a computation on the local variables M of $\mathcal{U}$ as shown in line 43 of Algorithm 1. In addition, each output transition $a \in O$ passes the parameter values $val(Y_a) = u_{\mathrm{A}}(val(X))$ to the corresponding input transitions. For each variable $y \in Y_{\mathrm{A}}$, we introduce a global variable g_y in $\mathcal{U}$ (line 15 in Algorithm 1) and write G_{A} for the set of all such variables. Then, we assign the global variables $G_a := u_a(val(M))$ in line 40 of Algorithm 1. In case of an input action $a \in I$, the valuation of X is updated based on its current valuation and parameters passed by the TIOA. In the TA $\mathcal{U}$, this is performed by assigning $u_a(M, G_a)$ to the local variables in line 34 of Algorithm 1. Concerning parameter passing for external actions in TIOA, it has to be noted that output channels are executed before the corresponding input channels, that is the update of the global variables G_a is performed instantaneously. As a consequence, all UPPAAL templates, where c_a is an input channel always access the latest valuation of G_a. Note that the definition of global variables for parameter passing appears in line 13 to 22 of Algorithm 1.

In summary, we obtain the following operation when executing transitions. If a transition t_a with an internal action $a \in H$ is taken, $\mathcal{U}$ takes the selfloop edge e_a and the local variables M are updated. A transition t_a with an output action $a \in O$ is taken if one of the corresponding edges e_{a_i}, $i = 1, \ldots, l_a$ in $\mathcal{U}$ fulfills its guard condition. In that case, the local variables

M as well as the global variables G_a for parameter passing are updated. In addition, the broadcast output channel c_a is triggered, leading to the instantaneous execution of the related input actions in other TIOA. That is, synchronization of transitions with a common name, as required by the TIOA definition is achieved in the proposed UPPAAL representation.

TIOA Trajectories

According to the TIOA definition, the "trajectories" part of $\mathcal{A}$ describes the evolution of analog variables over time between the occurrences of discrete transitions. For the conversion to UPPAAL, we now make use of Assumption 3 by realizing each analog variable $x \in X^{\mathrm{a}}$ by a clock variable $v_a \in K$ with time derivative $d(v_a) = 1$ in UPPAAL. It is further required to realize the **stop** conditions that are formulated as a predicate on the valuation of the variables X. According to Assumption 4, it holds that time is allowed to evolve in $\mathcal{A}$ as long as $w(val(X)) = false$ but time must stop (or a discrete transition must occur) as soon as $w(val(X)) = true$. Considering the representation $w(val(X)) = (w^1(val(X^{\mathrm{d}})) \wedge x^1 = v^1(val(X^{\mathrm{d}})) \vee \cdots \vee (w^m(val(X^{\mathrm{d}})) \wedge x^m = v^m(val(X^{\mathrm{d}}))$ in Assumption 4, we use the corresponding invariant l_idle.inv= $(\neg w^1(val(M)) \vee k_{x^1} \leq v^1(val(M)) \wedge \cdots \wedge (\neg w^m(val(M)) \vee k_{x^m} = v^m(val(M))$ in $\mathcal{U}$. The conversion of stop conditions into invariants of the idle location is performed in line 26 of Algorithm 1.

Initialization

According to Assumption 1, the variables X of $\mathcal{A}$ are initialized with unique values. In $\mathcal{U}$, this corresponds to initializing the local variables M. UPPAAL allows variable initialization under the automaton declarations while defining the variables. However this method is valid only for constant value initialization. However TIOA parameters, as well as constants, are used for variable initialization in TIOA. Hence we need a way of on the fly initialization for automata variables. To this end, we introduce a separate location l_init with the property *initial* in $\mathcal{U}$, and introduce and edge e_{init} such that $l\_init \overset{e_{init}}{\to} l\_idle$. We label l_init as committed such that the edge e_{init} is taken immediately as the first edge of the system execution. e_{init} has an empty guard and sync field. Only the update field executes the function $InitAutomaton(q_0)$ that assigns the initial state q_0. In Algorithm 1, l_init is defined in line 23, e_{init} is introduced in line 28 and the initialization is performed in line 29 with the function $InitAutomaton(q_0)$.

The UPPAAL templates given in Fig. 2.5 are produced by the TUConvert algorithm. The integer parameter *maxIt* of $UseOldInputA$ in Fig. 2.3 is passed as the parameter of the corresponding UPPAAL template. The discrete variables $\mathtt{maxoutA}^{\mathrm{d}}$ and $\mathtt{nextA}^{\mathrm{d}}$ correspond to the local variables *maxoutA* and *nextA*, whereas the analog variable $\mathtt{nowA}^{\mathrm{a}}$ is converted to the clock *nowA* in UPPAAL. In addition, the locations l_init and l_idle with the edge e_{init} for initialization are defined for each TIOA. The discrete transitions of the TIOA are defined by selfloop edges. In $UseOldInputA$, the edge e_{A} for action A receives the output channel e_{A}.sync = *a!* and the guard e_{A}.guard = $maxoutA > 0)\&\&(nowA == nextA)$ is used. The update $maxoutA = maxoutA - 1$ and $nextA = infty$ is performed in e_{A}.update = *Afunc().* Since B is an input action in $UseOldInputA$, the corresponding edge e_{B} obtains an input channel

e_{B}.sync = *B?* and has no guard condition. The update is performed with e_{B}.update = *Bfunc()* that realizes the effect of the transition B in $UseOldInputA$. The conversion of $UseOldInputB$ is analogous.

```
input  : $\mathcal{A}$
output: $\mathcal{U}$
1  $F := P_F$
2  // Global Structures
3  for $all\ p \in P_T$ do
4      typedef struct type($p$);
5  end
6  // Local Variables ($O$)
7  for $all\ x \in X^d$ do
8      type($x$) $v_x = x_0$;
9  end
10 for $all\ x \in X^a$ do
11     clock $k_x = x_0$;
12 end
13 // Global Variables ($G$)
14 for $all\ a \in O$ do
15     for $all\ parameters\ y \in Y_a\ passed\ by\ t_a$ do
16         type($y$) $g_y$;
17     end
18 end
19 // Channels
20 for $all\ a \in (I \cup O)$ do
21     broadcast channel $c_a$;
22 end
23 // Locations ($L$)
24 initial location $l\_init$;
25 location $l\_idle$;
26 $l\_idle$.inv = $(\neg w_1(val(M)) \vee (x_1 \leq v_1(val(M)))) \wedge (\neg w_2(val(M)) \vee (x_2 \leq v_2(val(M)))) \wedge \cdots$
27 // Edges ($E$)
28 $edge\ e_{init};\ with\ l\_init \overset{e_{init}}{\rightarrow} l\_idle$
29 $e_{init}.update = InitAutomaton(q_0)$;
30 for $all\ a \in S$ do
31     if $a \in I$ then
32         edge $e_a$ with $l\_idle \overset{e_a}{\rightarrow} l\_idle$
33         $e_a$.synch = $c_a$?;
34         $e_a$.update = $h_a(val(M), Y_a)$
35     else
36         for $i = 1, \ldots, k_a$ do
37             edge $e_{ai}$ st $l\_idle \overset{e_{ai}}{\rightarrow} l\_idle$
38             $e_{ai}$.guard = $f^{ai}(val(M))$;
39             if $a \in O$ then
40                 $G_{\text{A}} := u_{\text{A}}(val(M))$;
41                 $e_{ai}$.synch = $c_a$!;
42             end
43             $e_{ai}$.update = $g_a(val(M))$
44         end
45     end
46 end
```

Algorithm 1: *TUConvert* Algorithm

The TUConvert algorithm as outlined in Algorithm 1 is implemented in the form of the TU-Convert tool [63].

TUConvert receives an input xml file that is written in the TIOA syntax and produces an output xml file which can be opened in UPPAAL. The input xml file for the automata Use-OldInputA and UseOldInputB and corresponding output xml file with their definitions are given in Appendix A and Appendix B respectively. A screen-shot is shown in Fig. 5.2.

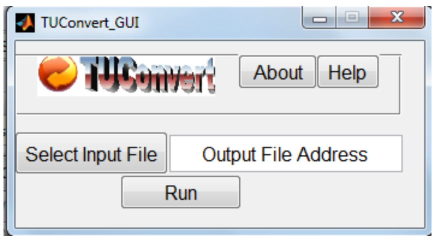


Figure 5.2: TUConvert Tool

5.4 Timed Automata Semantics in UPPAAL

This section defines the semantics for UPPAAL TA models that are used in Section 5.5 to establish the behavioral equivalence of distributed TIOA models and their UPPAAL TA models from TUConvert. We adapt the semantics in [37] to the special TA that result from TUConvert for this purpose. It is important to note that, our special case offers consistent semantics for dealing with shared variables which constitutes a novelty in the existing literature. In order to describe the semantics of a TA in UPPAAL, we introduce the notation $(l, v) \xrightarrow{e} (l', v')$ to denote a transition from location $l \in L$ and state $v \in val(V)$ to location $l' \in L$ and state $v' \in val(V)$ either by firing an edge $e \in E$ or by time passage $e \in \mathbb{R}$. In addition, for $v \in val(V)$ and $d \in \mathbb{R}$, we use the operator $v \oplus d$ [37] such that for each $y \in V$,

$$v \oplus d(y) = \begin{cases} v(y) + d & \text{if } y \in K \\ v(y) & \text{otherwise} \end{cases}$$

That is, only the valuation of clock variables in K is updated when time passes. Then, the semantics of a TA can be represented by a *labeled transition system* $\langle Y, y_0, \rightarrow \rangle$ with the states $Y \subseteq L \times val(V)$, the initial state $y_0 = (l_0, v_0)$ and the transition relation $\rightarrow$ such that

$$\begin{aligned} (l, v) \xrightarrow{d} (l, v \oplus d) \quad & \text{if } \forall d' \in \mathbb{R} : 0 \leq d' \leq d \Rightarrow I(l, v \oplus d') = true \\ (l, v) \xrightarrow{e} (l', v') \quad & \text{if } e = (l, g, a, r, l') \text{ with } g(v) = true, v' = r(v), I(l', v') = true. \end{aligned}$$

We finally define the execution semantics of a network of TA models as employed in this work. To this end, we consider a set $\mathcal{N} = \{\mathcal{U}_1, \ldots, \mathcal{U}_n\}$ of n models $\mathcal{U}_i = (F_i,\ L_i,\ l_{0,i},\ V_i,\ v_{0,i},\ C_i,\ A_i,\ I_i,\ E_i)$, $i = 1, \ldots, n$. Hereby, we note that the global variables $G \subseteq V_1 \cap \cdots \cap V_n$ are shared by all TAs and can in principle be modified by any TA. In the scope of this work, it is sufficient to assume that no global variable is a clock variable and the valuation of each global variable $g \in G$ can only be updated by a unique TA when taking edges with output channels.

Assumption 5 *Let $\mathcal{N}$ be a network of TA models with the global variables G. Then we assume that, for each $g \in G$, $g \notin K$ and there is a unique $\mathcal{U}_i \in \mathcal{N}$ that is permitted to update g when taking an edge with an output channel.*

This is a critical assumption that guarantees the correct message exchange between interacting automatons. This assumption guarantees that the global message carrying variables can only be updated before the corresponding output transition is triggered. Hence the automaton waiting for the corresponding input action transition will obtain the updated value.

Using Assumption 5, the execution semantics for $\mathcal{N}$ is given by the labeled transition system $\langle Y, y_0, \rightarrow\rangle$ with the variables $Y = L_1 \times \cdots \times L_n \times val(O_1) \times \cdots \times val(O_n) \times G$, the initial state $y_0 = (l_{0,1}, \ldots, l_{0,n}, m_{0,1}, \ldots, m_{0,n}, g_0)$ and the transition relation $\rightarrow$ as follows

Time Passage

$(l_1, \ldots, l_n, m_1, \ldots, m_n, g) \xrightarrow{d} (l_1, \ldots, l_n, m_1 \oplus d, \ldots, m_n \oplus d, g)$ if $d \in \mathbb{R}$ and $\forall d' \in \mathbb{R}$ and $i = 1, \ldots, n$: $0 \leq d' \leq d \Rightarrow I_i(l_i, m_i \oplus d', g) = true$.

Silent Transitions

$(l_1, \ldots, l_i, \ldots, l_n, m_1, \ldots, m_i, \ldots, m_n, g) \xrightarrow{\tau} (l_1, \ldots, l'_i, \ldots, l_n, m_1, \ldots, m'_i, \ldots, m_n, g')$
if $\exists e_i = (l_i, \tau, g_i, r_i, l'_i) \in E_i$ such that $g_i(m_i, g) = true, (m'_i, g') = r_i(m_i, g), I_i(l'_i, m'_i, g') = true$

Shared Transitions

$(l_1, \ldots, l_n, m_1, \ldots, m_n, g) \xrightarrow{c} (l'_1, \ldots, l'_n, m'_1, \ldots, m'_n, g')$ if $c \in C$,
$e_j = (l_j, c!, g_j, r_j, l'_j) \in E_j$ for one j with $g_j(m_j, g) = true, (m'_j, g') = r_j(m_j, g), I_j(l'_j, m'_j, g') = true$
$e_i = (l_i, c?, g_i, r_i, l'_i) \in E_i$ for $1 \leq i \leq n$ with $g_i(m_i, g) = true, (m'_i, g) = r_i(m_i, g), I_i(l'_i, m'_i, g) = true$
$l'_k = l_k$ and $m'_k = m_k$ for all remaining $1 \leq k \leq n$.

That is, time passage requires that all automata fulfill their respective invariant while clock variable valuations increase and the remaining variable valuations do not change. A silent transition can occur in a single automaton if there is a corresponding edge whose guard is fulfilled and such that the variable valuation meets the invariant of the target location after the update.

Shared transitions require the firing of one edge with an output channel $c!$ in one of the automata, whereby all automata are synchronized on edges with the input channel $c?$ if their guard is fulfilled and the updated variable valuation meets the respective invariant. Hereby, the global variables g are only updated by the automaton with the edge with the output channel according to Assumption 5. All remaining automata remain in the same location and do not update their local variable valuations.

5.5 Formal Results

The main objective of TUConvert is the conversion of a network of TIOA to a network of UPPAAL automata models for formal verification. To this end, it is essential that both models represent the same behavior for the completeness of the conversion procedure. Consider a composition of TIOA models given by $\mathcal{A} = ||_{i=1}^{n}\mathcal{A}_i$ and assume that $\mathcal{N}$ is the network of UPPAAL automata models resulting from the application of TUConvert to $\mathcal{A}_1, \dots, \mathcal{A}_n$. In this section, we show that both models exhibit an equivalent behavior in the sense that

1. every execution in $\mathcal{A}$ has a correspondence in $\mathcal{N}$
2. every possible path in $\mathcal{N}$ has a corresponding execution in $\mathcal{A}$

We formalize this important fact in the following theorem.

Theorem 3 *Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be TIOA models and $\mathcal{U}_1, \dots, \mathcal{U}_n$ be the corresponding UPPAAL automata models obtained by Algorithm 1. Write $\mathcal{A} = ||_{i=1}^{n}\mathcal{A}_i$ and $\mathcal{N}$ for the network of UPPAAL automata models.*

1. *Let $\tau_0 \text{A}_1 \tau_1 \cdots \text{A}_k \tau_k$ be an execution in $\mathcal{A}$. Then there exist states $y_j = (l\_idle, \dots, l\_idle, m_{1,j}, \dots, m_{n,j}, g_j) \in val(Y)$ for each $j = 1, \dots, k$ such that*

   (a) $\forall d, 0 \leq d \leq \tau_j.ltime - \tau_j.ftime$:

   $$(l\_idle, \dots, l\_idle, m_{1,j}, \dots, m_{n,j}, g_j) \xrightarrow{d} (l\_idle, \dots, l\_idle, m_{1,j} \oplus d, \dots, m_{n,j} \oplus d, g_j)$$

   exists in $\mathcal{N}$ and

   $$\tau_j(\tau_j.ftime + d) \lceil X_i = m_{i,j} \oplus d$$

   for all $i = 1, \dots, n$.

   (b) $$(l\_idle, \dots, l\_idle, m_{1,j} \oplus d, \dots, m_{n,j} \oplus d, g_j) \xrightarrow{\text{A}_{j+1}} (l\_idle, \dots, l\_idle, m_{1,j+1}, \dots, m_{n,j+1}, g_{j+1})$$
   exists in $\mathcal{N}$ and

   $$\tau_{j+1}.fval \lceil X_i = m_{i,j+1}$$

   for each $i = 1, \dots, n$.

2. *Assume that there exist states $y_j = (l\_idle, \dots, l\_idle, m_{1,j}, \dots, m_{n,j}, g_j) \in val(Y)$, times $d_0, \dots, d_{k-1} \in \mathbb{R}$ and actions $\text{A}_1, \dots, \text{A}_k \in A$ such that $\forall j = 1, \dots, k-1$,*

   (a) $\forall d, 0 \leq d \leq d_j$, $(l\_idle, \dots, l\_idle, m_{1,j} \oplus d, \dots, m_{n,j} \oplus d, g_j) \in val(Y)$

   (b) $(l\_idle, \dots, l\_idle, m_{1,j} \oplus d_j, \dots, m_{n,j} \oplus d_j, g_j) \xrightarrow{\text{A}_{j+1}} (l\_idle, \dots, l\_idle, m_{1,j+1}, \dots, m_{n,j+1}, g_{j+1})$

   Then, there exist $\tau_0, \dots, \tau_{k-1} \in \mathcal{T}$ such that

   $$\tau_0 \text{A}_1 \tau_1 \cdots \tau_{k-1} \text{A}_k$$

   is an execution in $\mathcal{A}$.

In order to prove Theorem 3, we first state two lemmas. Lemma 6 is concerned with the correspondence of trajectories in TIOA and time passage in UPPAAL automata models generated by TUConvert. Lemma 7 establishes the relation between the occurrence of actions in TIOA and the firing of edges in the corresponding UPPAAL automata models.

Lemma 6 *Consider the compatible TIOA $\mathcal{A}_1, \ldots, \mathcal{A}_n$ with $\mathcal{A} = ||_{i=1}^{n} \mathcal{A}_i$ and the corresponding network of UPPAAL automata models $\mathcal{N} = \{\mathcal{U}_1, \ldots, \mathcal{U}_n\}$ that is generated with Algorithm 1. Assume $x = (x_1, \ldots, x_n) \in Q$ is a state of $\mathcal{A}$ and $(l_1, \ldots, l_n, m_1, \ldots, m_n, g) \in Y$ is a state of $\mathcal{N}$ such that $x_i = m_i$ for all $i = 1, \ldots, n$. Let $\tau \in T$ be a trajectory with $\tau.fval = x$.*

Then, τ is an execution fragment in $\mathcal{A}$ if and only if $\forall d$ with $0 \leq d \leq \tau.ltime - \tau.ftime$, the transition $(l_1, \ldots, l_n, m_1, \ldots, m_n, g) \xrightarrow{d} (l_1, \ldots, l_n, m_1 \oplus d, \ldots, m_n \oplus d, g)$ exists in $\mathcal{N}$ and $\tau(\tau.ftime + d)\lceil X_i = m_i \oplus d$ for all $i = 1, \ldots, n$.

Proof 9 *Assume that τ is an execution fragment in $\mathcal{A}$*

$\Leftrightarrow \forall i = 1, \ldots, n, \tau.fval\lceil X_i \in Q_i$ *and* $\tau \downarrow X_i \in T_i$

$\Leftrightarrow$ *no stop condition is fulfilled*

$\Leftrightarrow \forall t, \tau.ftime \leq t \leq \tau.ltime, w_i(\tau(t)\lceil X_i) = false$

We note that it holds that $\tau(t) = \tau.fval \oplus d$ for all $t = \tau.ftime + d$ because of Assumption 3. Hence $m_i \oplus d = \tau.fval \oplus d\lceil X_i$ for all d, $0 \leq d \leq \tau.ltime - \tau.ftime$.

$\Leftrightarrow \forall d, 0 \leq d \leq \tau.ltime - \tau.ftime, l\_idle.inv(m_i \oplus d) = true$ *(see line 26 in TUConvert)*

$\Leftrightarrow \forall d, 0 \leq d \leq \tau.ltime - \tau.ftime, (l_1, \ldots, l_n, m_1, \ldots, m_n, g) \xrightarrow{d} (l_1, \ldots, l_n, m_1 \oplus d, \ldots, m_n \oplus d, g)$
exists in $\mathcal{N}$ and $\tau(\tau.ftime + d)\lceil X_i = m_i \oplus d$ for all $i = 1, \ldots, n$.

Lemma 7 *Consider the compatible TIOA $\mathcal{A}_1, \ldots, \mathcal{A}_n$ with $\mathcal{A} = ||_{i=1}^{n} \mathcal{A}_i$ and the corresponding network of UPPAAL automata models $\mathcal{N} = \{\mathcal{U}_1, \ldots, \mathcal{U}_n\}$ that is generated with Algorithm 1. Assume $x \in Q$ is a state of $\mathcal{A}$ and $(l\_idle, \ldots, l\_idle, m_1, \ldots, m_n, g) \in Y$ is a state of $\mathcal{N}$ such that $x\lceil X_i = m_i$ for all $i = 1, \ldots, n$. Let $\text{A} \in S$ be an action in $\mathcal{A}$.*

Then, $x \xrightarrow{\text{A}} x'$ is a transition in $\mathcal{A}$ with $x' \in Q$ if and only if there exists a state $(l\_idle, \ldots, l\_idle, m_1', \ldots, m_n', g')$ such that the transition

$$(l\_idle, \ldots, l\_idle, m_1, \ldots, m_n, g) \xrightarrow{\text{A}} (l\_idle, \ldots, l\_idle, m_1', \ldots, m_n', g')$$

exists in $\mathcal{N}$.

Moreover, in that case, it holds for all $i = 1, \ldots, n$ that $x'\lceil X_i = m_i'$.

Proof 10 *Let $x \xrightarrow{\text{A}} x'$ be a transition in $\mathcal{A}$. That is, the action A is enabled at the state x of*

$\mathcal{A}$.

$\Leftrightarrow$ *$\exists i$ such that* $\text{A} \in O_i$ *or* $\text{A} \in H_i$, *and the precondition* $f_i^{\text{A}}(x\lceil X_i) = true$.

For all remaining i such that $\text{A} \in S_i$, *it holds that* $\text{A} \in I_i$ *since* $\mathcal{A}_1, \ldots, \mathcal{A}_n$ *are compatible*

$\Leftrightarrow$ *for i such that* $\text{A} \in O_i$ *or* $\text{A} \in H_i$, *there is an edge* e_{A} *with* $l\_idle \xrightarrow{e_{\text{A}}} l\_idle$ *and*

$e_{\text{A}}.synch = c_{\text{A}}!$ *and* $e_{\text{A}}.guard(m_i) = true$ *since* $m_i = x\lceil X_i$ *(see line 32, 41, 38 in TUConvert).*

Then, $g^{\text{A}} = h^{\text{A}}(m_i)$ *is set if* $\text{A} \in O_i$, *whereas* g^{A} *is unused if* $\text{A} \in H_i$.

Furthermore, $\forall i = 1, \ldots, n, e_a.update(m_i, g\text{A}) = e_a.update(x_i, g^{\text{A}}) = x'\lceil X_i = m_i'$

(see line 34, 40, 43 in TUConvert) and $l\_idle.inv(x'\lceil X_i) = true$ *because of Assumption 3 and line 26 in TUConvert*

$\Leftrightarrow (l\_idle, \ldots, l\_idle, m_1, \ldots, m_n, g) \xrightarrow{\text{A}} (l\_idle, \ldots, l\_idle, m_1', \ldots, m_n', g')$ *exists in* $\mathcal{N}$

and $x'\lceil X_i = m_i'$ *for all* $i = 1, \ldots, n$.

Now it is possible to prove Theorem 3.

Proof 11 *1. We prove the assertion by induction. Initially, we consider the trajectory* τ_0 *and consider* $x_0 = (x_{1,0}, \ldots, x_{n,0}) = \tau_0.fval$. *Since* $\tau_0.fval = q_0$, *it holds for each* $i = 1, \ldots, n$ *that* $x_{i,0} = q_{0,i}$ *by definition of the TIOA composition and by Assumption 1. Let* $y = (l_1, \ldots, l_n, m_{0,1}, \ldots, m_{0,n}, g_0)$ *be the initial state of* $\mathcal{N}$. *Then, it holds that* $l_i = l\_init$ *by line 23 of TUConvert. Since the edges* $l\_init \xrightarrow{e_{init}} l\_idle$ *are committed in each* $\mathcal{U}_i$, $i = 1, \ldots, n$ *with line 28 of TUConvert, it is also true that each* $\mathcal{U}_i$ *transitions to idle location while initializing the state variables (line 29) immediately at time* 0. *Hence, at time* 0, *the state of* $\mathcal{N}$ *assumes the value* $y_0 = (l\_idle, \ldots, l\_idle, m_{0,1}, \ldots, m_{0,n}, g_0)$, *whereby* $m_{0,i} = x_{i,0} = q_{i,0}$ *for all* $i = 1, \ldots, n$. *That is, at time* 0, *we have the state x of* $\mathcal{A}$ *and the state* y_0 *of* $\mathcal{N}$ *such that* $m_{i,0} = x_{i,0}$ *for all* $i = 1, \ldots, n$.

Then, it is true that $m_0 = \tau_0(0)$. *Since* τ_0 *is an execution in* $\mathcal{A}$ *and* $\tau_0.fval = x$, *Lemma 6 implies that 1.(a) in Theorem 3 is fulfilled.*

Next, we assume that the conditions in Theorem 3 1. are fulfilled for $\tau_0\text{A}_1\tau_1 \cdots \text{A}_j\tau_j$. *We need to show that these conditions are also true for* $\tau_0\text{A}_1\tau_1 \cdots \text{A}_j\tau_j\text{A}_{j+1}\tau_{j+1}$.

We first note that, for $d_j = \tau_j.ltime - \tau_j.ftime$, $\tau_j(d_j)\lceil X_i = m_{j,i} \oplus d_j$ *for all* $i = 1, \ldots, n$ *according to the induction assumption. Considering that* A_{j+1} *is enabled at state* m_j *in* $\mathcal{A}$, *Lemma 7 implies that there exist* $m_{i,j+1}$ *for* $i = 1, \ldots, n$ *and* $g_{j+1} \in G$ *such that* $(l\_idle, \ldots, l\_idle, m_{1,j}, \ldots, m_{n,j}, g_j) \xrightarrow{\text{A}_{j+1}} (l\_idle, \ldots, l\_idle, m_{1,j+1}, \ldots, m_{n,j+1}, g_{j+1})$ *and* $m_{i,j+1} = \tau_0\text{A}_1\tau_1 \cdots \text{A}_{j+1}\tau_{j+1}.fval\lceil X_i$ *for all* $i = 1, \ldots, n$. *Since,* $\tau_0\text{A}_1\tau_1 \cdots \text{A}_{j+1}\tau_{j+1}$ *is an execution in* $\mathcal{A}$, *the same argument as before implies that for all d,* $0 \leq d \leq \tau_{j+1}.ltime - \tau_{j+1}.ftime$, $(l\_idle, \ldots, l\_idle, m_{1,j+1}, \ldots, m_{n,j+1}, g_{j+1}) \xrightarrow{d} (l\_idle, \ldots, l\_idle, m_{1,j+1} \oplus d, \ldots, m_{n,j+1} \oplus d, g_{j+1})$ *and* $m_{i,j+1} \oplus d = \tau_{j+1}(\tau_{j+1}.ftime + d)\lceil X_i$ *for all* $i = 1, \ldots, n$. *This completes the proof of 1.*

2. *The proof of this item follows the same argument as in item 1. in reverse direction and is omitted for brevity.*

To sum up Theorem 3 guarantees that each execution fragment and state of an input TIOA has a corresponding equivalent in the output UPPAAL model. Hence we say that, TUConvert defines a complete conversion procedure. Moreover the correspondance relation between the input TIOA and output UPPAAL models has a one to one mapping characteristics. While TUConvert generates a unique output for any correct input, the conversion is in general not unique.

5.6 Modeling Case Study: Distributed Real-Time Protocol

We now apply the *TUConvert* algorithm to the distributed real-time control suite for industrial control systems given in Section 4.2 in the TIOA formalism. The framework has a layered architecture over a shared medium broadcast channel as shown in Figure 4.3, whereby the *Interface Layer (IL)*, *Coordination Layer (CL)* and *Dependability Plane (DP)* are the main layers. This section gives the UPPAAL templates of the framework layers by using Algorithm 1. The protocol family including the dependability plane are verified by using the UPPAAL tool suite. The verification details are given in Chapter 6

5.6.1 Shared Medium Broadcast Network

The UPPAAL template of the shared medium broadcast network, which is generated by Algorithm 1, is given in Fig. 5.3.

The SM template defines the variables *X* of the corresponding TIOA model as local variables. *l_init* and *l_idle* are the locations of the template. *l_init* is defined as the committed initial location of the template Hence, it is guaranteed that the template operation is started from this location by an outgoing edge transition handling the initial value assignment of the local variables. *l_idle* is used as the location where time passage is allowed, and the template is designed to wait at the *l_idle* location for an input channel trigger *IL2SM?* or for the time instant to trigger the output channel *SM2ILDL!*.

Separate loopback edges are defined in the UPPAAL template for each of the possible actions IL2SM and SM2ILDL. The transition preconditions and effects of the corresponding TIOA transitions in Fig. 4.4 are used to define the guard and update conditions of the corresponding edges respectively. The shared-medium broadcast network shares the actions SM2ILDL and IL2SM with the interface layers of the distributed nodes connected to the shared-medium broadcast network. Hence *SM2ILDL* and *IL2SM* are defined as global broadcast channels.

Note that the defined channels appear in the synch fields of the corresponding edges in the UPPAAL template in Fig. 5.3. The functions IL2SMfunc() and SM2ILDLfunc() implements the "eff" fields of the corresponding discrete transitions IL2SM and SM2ILDL given in Fig. 4.4 respectively.

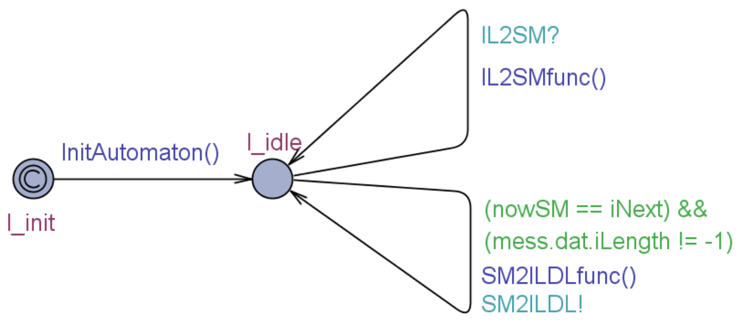


Figure 5.3: UPPAAL template for the shared-medium broadcast network.

5.6.2 Interface Layer

As described in Section 4.1.3, IL provides time-slotted access to the underlying shared-medium broadcast channel. Moreover it handles the slot allocation for non-real-time messages of different network nodes. The TIOA model for the IL is given in Figure 4.5. Note that IL is designed such that an instance of the layer runs on each node that participates in the communication. Because of this reason, we specify entities that are specific to each instance (node) by the index i. In order to perform the conversion to UPPAAL, we first confirm that all assumptions in Section 5.2 are fulfilled. The initial state is unique (Assumption 1), the single analog variable $\mathtt{now}_i^{\mathrm{a}}$ evolves with $d(\mathtt{now}_i^{\mathrm{a}}) = 1$ and the only operation on $\mathtt{now}_i^{\mathrm{a}}$ is the reset (Assumption 3). Considering that the types M, Q, A_{IL} and H_{IL} are represented by *structs* in the C programming language, also Assumption 2 holds. Finally Assumption 4 holds for IL since $\mathtt{now}_i^{\mathrm{a}}$ is the only analog variable and hence there is only one disjunct in the transition preconditions. We now construct the UPPAAL template of IL.

TIOA Parameters: The values $dSlot$, t_{IL0}, t_{IL1}, t_{IL2}, t_{IL3} and $InitIL$ are passed as parameters F of $\mathcal{U}$. In addition, global structures for the types M, Q, A_{IL} and H_{IL} are defined.

TIOA Variables: All variables X in the "states" part of the IL TIOA are defined as local variables L of $\mathcal{U}$ according to their data type. The only real-valued variable $\mathtt{now}_i^{\mathrm{a}}$ is converted as the clock variable.

TIOA Signature: The external actions of the IL TIOA are $\textsc{il2sm}_i$, $\textsc{sm2ildl}$, $\textsc{cl2ilrt}_i$, $\textsc{ap2ilNrt}_i$, $\textsc{il2apNrt}_i$, $\textsc{il2clrt}_i$, $\textsc{reqrt}_i$, $\textsc{updvil}_i$, $\textsc{sendres}_i$ and $\textsc{rback}_i$. For each such action, a broadcast channel is introduced (for example, $c_{\textsc{cl2ilrt}}$ for the action $\textsc{cl2ilrt}$). Moreover, a global variable is generated for each parameter, that is passed, when taking the respective transition. For example, considering the output action $\textsc{il2clrt}(m : M)_i$, there is a global variable of type M.

TIOA Transitions: The UPPAAL template $\mathcal{U}$ is at the location l_{idle}, where time can pass. Each transition in IL TIOA is represented by a loopback edge, that is originating from the location l_idle in the UPPAAL automata.

We discuss two examples. The edge e_{SM2ILDL} for the input transition SM2ILDL is defined without guard condition, since there is no precondition in the IL TIOA. The edge is synchronized with edges in other TIOA via the broadcast channel c_{SM2ILDL} that is used as input channel $c_{\text{SM2ILDL}}?$ in the sync field of e_{SM2ILDL}. Finally, the pseudo code in the "eff" part of the IL TIOA is directly translated to the syntax of $\mathcal{U}$. Now consider the internal transition UPDATE. Here, the pseudo code in the "pre" part is directly translated to the UPPAAL syntax in the guard field of the edge e_{UPDATE}. Similarly, the pseudo code in the "eff" part appears as "update" field in the UPPAAL template. Finally, the "sync" field of the edge e_{UPDATE} remains empty.

TIOA Trajectories: The invariant of l_{idle} is deduced from the stop condition in the "trajectories" part of $\mathcal{A}$. Consider for example, the expression $\texttt{reqIL}_i^{\text{d}} \wedge (\texttt{now}_i^{\text{d}} = t_{IL0})$ in the stop condition of the IL TIOA. This expression is converted in the invariant $\neg\texttt{reqIL}_i^{\text{d}} \vee (\texttt{now}_i^{\text{d}} \leq t_{IL0})$ in location l_{idle}.

Initialization: Initialization of the variables V is performed instantaneously in the committed initial location l_{init}. The variables V are initialized according to their initialization in the "states" part of the IL TIOA. For example, $\texttt{RTIL}_i^{\text{d}}$ is initialized as false.

The derived UPPAAL model of IL is given in Fig. 5.4.

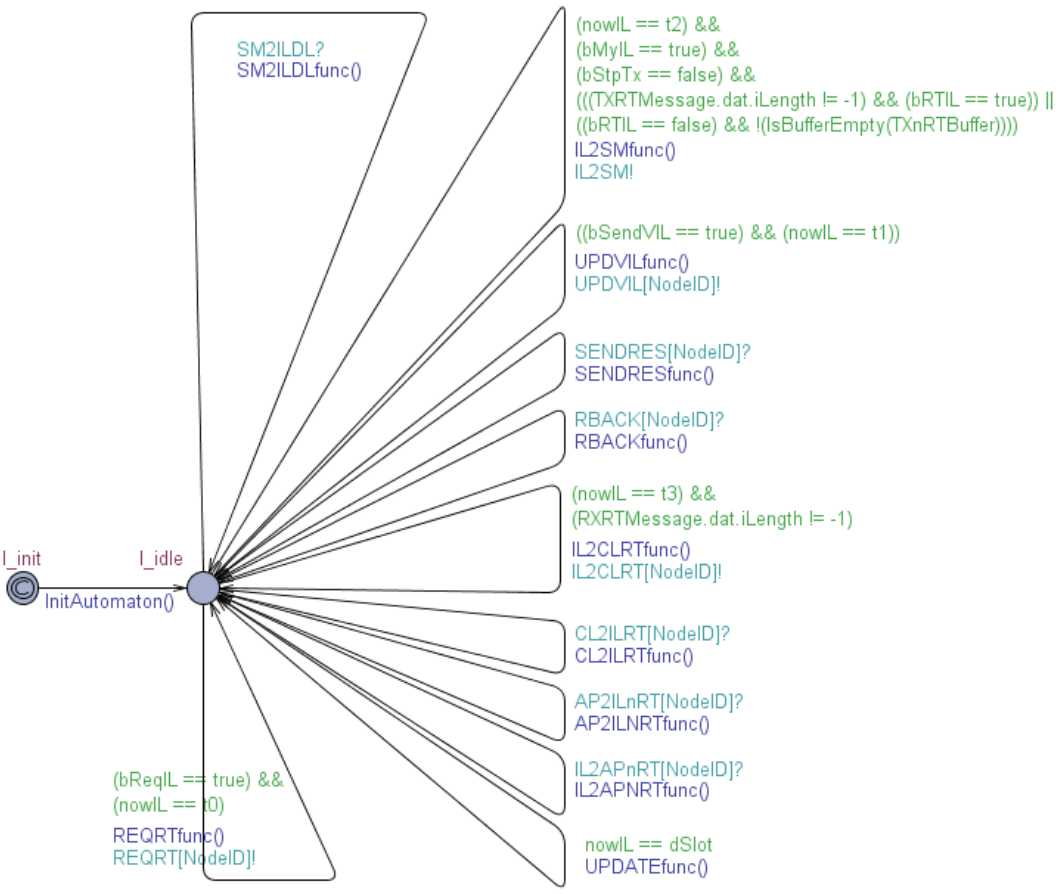


Figure 5.4: Interface Layer Implementation In UPPAAL

5.6.3 Coordination Layer

The CL of the proposed protocol suite is responsible for the communication coordination of the real-time slots provided by the IL. CL has communication interfaces with IL, DP and the connected control applications (see Figure 4.3). Loosely speaking, CL uses data provided from the control application during run-time in order to uniquely assign real-time slots to the nodes participating in the communication, hence, avoiding collisions on the shared medium broadcast channel. The CL TIOA is given in Figure 4.6. Similar to the IL, it has to be noted that an instance of the CL is running on each node, specified by the index i.

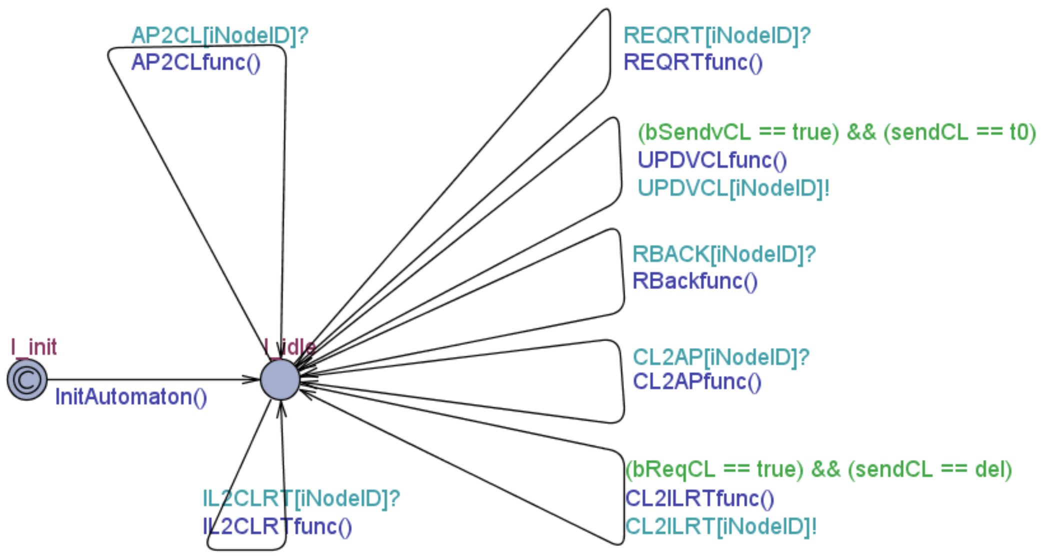


Figure 5.5: Coordination Layer Implementation In UPPAAL

The corresponding UPPAAL template of the CL is given in Figure 5.5. The model is produced by using the *TUConvert* algorithm. Since the procedure is analogous to the IL conversion, we only point out special features of the CL.

TIOA Variables: One clock variable is introduced for the analog variable $\mathtt{send}_i^{\mathrm{a}}$ of the CL TIOA. All other variables are defined according to their data types.

TIOA Signature: The external actions $\textsc{il2clrt}_i$, $\textsc{cl2ilrt}_i$, $\textsc{reqrt}_i$, $\textsc{ap2cl}_i$, $\textsc{cl2ap}_i$, $\textsc{updvcl}_i$ and $\textsc{rback}_i$ are used as communication interfaces of CL with IL, DP and the control application running on top. These external actions are used as common broadcast channels between CL and interface automatons. The parameters $\mathtt{RTCL}_i^{\mathrm{d}}$, $\mathtt{myCL}_i^{\mathrm{d}}$ and m, that are passed by the output action $\textsc{cl2ilrt}_i$ are written to global variables in CL template.

TIOA Transitions: Regarding the transitions of the CL TIOA, we emphasize the synchronization of shared actions. The "sync" field of the loopback edge $e_{\textsc{cl2ilrt}_i}$ is used to trigger an output channel $c_{\textsc{cl2ilrt}_i}!$, whereas the "sync" field for the corresponding edge $e_{\textsc{cl2ilrt}_i}$ in the IL model is an input channel $c_{\textsc{cl2ilrt}_i}?$. An analogous correspondence is observed for the input actions $\textsc{il2clrt}_i$ and $\textsc{reqrt}_i$. We also determine the invariant of the location l_{idle} as the negation of the stop condition: $\neg\mathtt{reqCL}_i^{\mathrm{d}} \vee (\mathtt{send}_i^{\mathrm{a}} \leq del_i)$.

The remaining steps of the conversion procedure are analogous to the detailed description in Section 5.6.2. Hence, we refrain from repeating these steps.

5.6.4 Dependability Plane

The dependability plane, as the main subject of Chapter 3, is responsible for fullfiling the dependability related requirements of the proposed framework. DP, as given in Figure 4.3, has communication interfaces with IL, CL and the shared medium broadcast channel (SM).

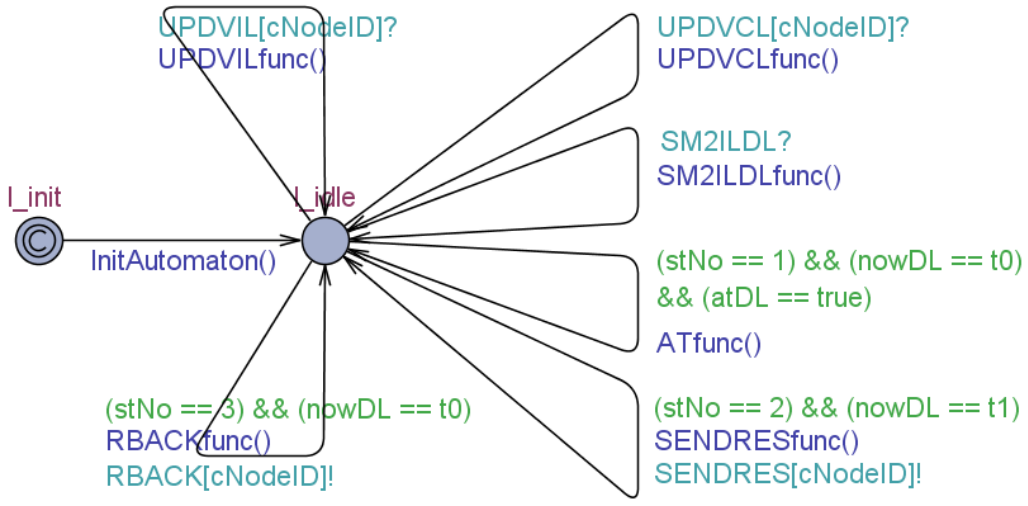


Figure 5.6: Dependability Plane Implementation In UPPAAL

DP uses data provided from the connected framework layers IL,CL and the information padded to the application message from SM to make a consistency check in order to determine a node level fault condition. DP TIOA is given in Figure 3.7. Similar to the IL and CL, it has to be noted that an instance of the DP is running on each node, specified by the index *i*.

The corresponding UPPAAL template of the DP is given in Figure 5.6. The model is produced by using the *TUConvert* algorithm. Since the procedure is analogous to the IL and CL conversion, we only point out special features of the DP.

TIOA Variables: One clock variable is introduced for the analog variable $\mathtt{now}_i^{\mathrm{a}}$ of the DP TIOA. All other variables are defined according to their data types.

TIOA Signature: The external actions $\textsc{updvil}_i$, $\textsc{updvcl}_i$, $\textsc{sendres}_i$, $\textsc{rback}_i$ and $\textsc{sm2ildl}_i$ are used as communication interfaces of DP with IL, CL and SM. These external actions are used as common broadcast channels between DP and interface automatons.

TIOA Transitions: DP uses shared actions with the interface automatons in order to have a synchronized operation. Global variables are used for parameter passing between DP and IL,CL or SM. The conversion procedure for DP model is similar to the CL and IL models as described in Section 5.6.2 and Section 5.6.3 respectively.

CHAPTER 6

SIMULATION AND VERIFICATION OF TIOA BASED MODELS IN UPPAAL

Formal verification generally deals with the problem of whether a system design logically implies its specification. [25] Model-checking is a powerful technique to formally verify a system, whose behavioral model is given as a concurrent state-graph model. Model checking is done by comparing the behavioral model of the system with a temporal logic formula [26] and has several important verification advantages such as:

- Shortening the formal verification process via automation,
- Reducing the complexity of the verification process via partial function checking,
- Enabling the user to directly express the properties needed to verify a sequential system behavior via the property specification logic. [27]

There are two main types of specifications that attract the academia and industry in the concept of formal verification. The first one implies that "bad things will never happen", while the second implies that "good things will happen" [35]. Timed Computation Tree Logic (TCTL) [26] like branching temporal logic formulates these two types of specifications with certain keywords. These properties are called as the *safety* and *reachability* properties within the TIOA and TCTL contexts.

TCTL is an extension of the computation tree logic with specific timing constraints. The formulae of TCTL are built from state predicates, by boolean connectives and temporal operators [27]. TCTL uses two common state formulae operators <> and [] meaning *paths* and *states* respectively. For path formulae *A* operator is used to imply *for all* and *E* operator is used to imply *there exist*. [64]. Example TCTL formulas are given in 6.1 and 6.2 with their meanings.

$$A\,[]\;p; \%\textit{For all states p holds} \tag{6.1}$$

$$E\;<>\;q; \%\textit{There exists a path where}\ \mathrm{q}\ \textit{holds} \tag{6.2}$$

While verifying distributed real-time systems, modeled in TIOA syntax, the system verification can be accomplished via verifying the individual automatons as well as verifying the whole system. This idea is based on the composition property of Timed Input/Output Automaton framework. UPPAAL uses a subset of TCTL allowing to write queries for both safety and reachability properties of TIOA systems.

In Section 6.1 verification of the critical safety and reachability properties of Timed Input/Output Automatons are described over the framework layers given in Section 4.1.

6.1 Formal Verification of D^{3}RIP Framework

6.1.1 Verification of Safety Properties

Safety properties describe the automaton specifications that should be satisfied for all conditions. The most widely used safety property, which is essential for all real-time system automatons is "Deadlock Freedom".

"Deadlock Freedom", as the name implies is the safety property of an automaton that guarantees the full-time operation without any action-locks. In other words the property guarantees the automaton's ability to make event-triggered transitions.

"Deadlock Freedom" can be verified by using the TCTL query 6.3.

$$A\,[]\ not\ deadlock; \tag{6.3}$$

Query 6.3 defines the necessary and sufficient condition for dead-lock freedom of an automaton and it is directly implementable in UPPAAL.

Since safety properties are independent of the execution fragment of the automaton under consideration, full coverage is a prerequisite for the verification of safety properties. UPPAAL calculates the coverage percentile of the system under discussion. Hence we use this property to guarantee full state coverage in the system operation. (i.e *number of visited locations* = *number of total locations*)

Safety properties are not restricted to just "Deadlock Freedom". The rules and state transition preconditions which are not dependent on the execution fragment of the automatons are also considered as Safety Properties. Safety properties of the D^{3}RIP framework layers are verified in UPPAAL.

6.1.2 Verification of Reachability Properties

Reachability properties define the automaton specifications which can be satisfied by at least one reachable state within the automaton definition. Reachability properties are mostly used in the design phase of the automaton. They describe the automaton specifications that are satisfied in at least one case. Form of the TCTL queries defining the reachability properties are given in Query 6.4.

$$E <> \ q; \%\textit{There exists a path that property q is satisfied} \tag{6.4}$$

Reachability properties are used mainly in the automaton design phase. Reachability properties of the D^{3}RIP framework layers are verified in UPPAAL.

6.1.3 Verification Results of D^{3}RIP Framework

UPPAAL can be used to model systems composed of multiple automata. Deadlock freedom, time synchronization and content synchronization like safety properties and reachability properties are the most critical properties of such systems. In our case, we want to verify these properties for the parallel operation of our UPPAAL templates.

We have designed a simple application layer template to run over the designed nodes. The application layer template, shown in Fig. 6.1, sends and receives messages in a ring topology. We have set two different networks composed of 2 and 3 nodes respectively. The communication starts with first node by transmitting the first message to the second node. Message transmission continues in a ring topology.

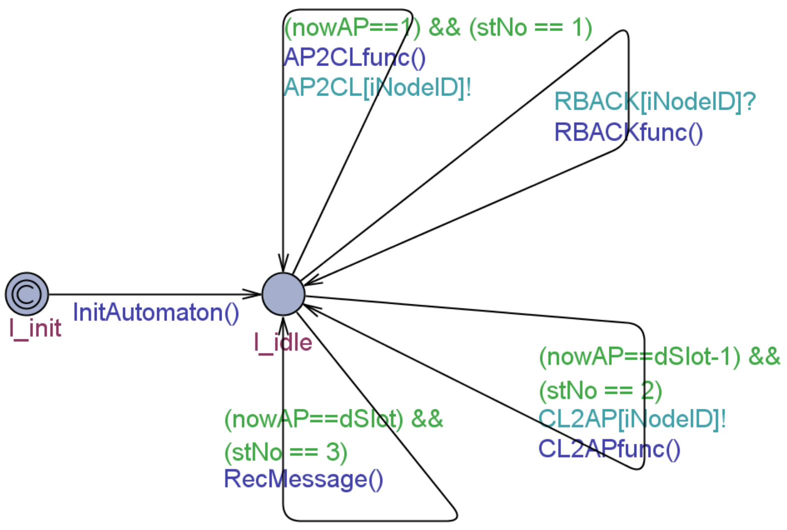


Figure 6.1: Application Layer Template

The queries that are used in the verification process are given in Table 6.1. The first query is used to verify the safety property stating that the system is deadlock free. The second query is used to verify an example reachability property of the network via checking the state reachability of *l_idle* in SM. The queries from 3 to 4 are used to verify the time synchronized operation of the distributed nodes by checking whether the IL automata of each node issue a message request to the CL automata at the same time.

Table6.1: Verification Queries

Query Number	Query
1	$A\Box$ *not deadlock*
2	$E\Diamond$ $SM.l\_idle$
3	$A\Box$ $(IL1.bReqIL == true)$ && $(IL1.nowIL == t1)$ *imply* $IL2.nowIL == t1$
4	$A\Box$ $(IL1.bReqIL == true)$ && $(IL1.nowIL == t1)$ *imply* $IL3.nowIL == t1$
5	$A\Box$ $(CL1.vCL.iCnt == CL2.vCL.iCnt)$ && $(CL1.sendCL == CL2.sendCL)$ *imply* $(CL1.vCL == CL2.vCL)$
6	$A\Box$ $(CL1.vCL.iCnt == CL3.vCL.iCnt)$ && $(CL1.sendCL == CL3.sendCL)$ *imply* $(CL1.vCL == CL3.vCL)$
7	$A\Box$ $(IL1.nowIL == IL2.nowIL)$ && $(IL1.vIL.iCnt == IL2.vIL.iCnt)$ *imply* $(IL1.vIL == IL2.vIL)$
8	$A\Box$ $(IL1.nowIL == IL3.nowIL)$ && $(IL1.vIL.iCnt == IL3.vIL.iCnt)$ *imply* $(IL1.vIL == IL3.vIL)$

Since the message transmitting node is decided on the fly, the variable contents that are used in the decision process should be the same at each node in the network. The queries from 5 to 8 are written to check the content synchronization of the distributed nodes via checking contents of the variables *vIL* and *vCL* of the network layers.

The queries given in Table 6.1 are applied to 2 different network configurations. In the first configuration the network is composed of 2 nodes that are sending real-time messages to each other. In the second configuration we inserted 1 additional node into the network such that 3 nodes send and receive messages in a ring topology respectively. We successfully verified the described queries with a positive result using UPPAAL on a laptop computer having a 2 GHz Intel T5750 dual core processor with 3 GB RAM. We have used a 32 bit Windows-7 operating system. The verification times and the peak memory usages of UPPAAL are given in Table 6.2 in order to show the complexity of the system under consideration.

As seen from Table 6.2, the fastest verification with minimum resource consumption is achieved for the reachability properties. On the other hand the verification of the safety properties, takes much longer verification times with increasing resource consumption.

The second important result that is observed from Table 6.2 is the increasing verification time and memory usage in case of an increasing number of nodes in the network configuration. The

Table6.2: Verification Results

Query Number	Number of Nodes	Verification Time	Peak Resident Memory Usage	Peak Virtual Memory Usage
1	2	438.144s	907,268KB	1,823,396KB
1	3	1436.730s	1,234,721KB	2,357,261KB
2	2	0.281s	21,124KB	59,200KB
2	3	0.39s	50,744KB	120,892KB
3	2	272.011s	875,621KB	1,801,138KB
3	3	652.311	987,288KB	2,030,728KB
4	3	652.118	987,188KB	2,029,123KB
5	2	274.063s	902,648KB	1,813,900KB
5	3	1669s	63,320KB	136,792KB
6	3	1654s	63,904KB	137,956KB
7	2	273.313s	897,912KB	1,804,924KB
7	3	647.30s	981,204KB	2,024,987KB
8	3	646.123s	980,998KB	2,023,789KB

verification times and memory usages seem to increase in quadratically in the node number. For instance, the verification of deadlock freedom for a 3 node network takes nearly 1437 seconds with a 2.36 GB peak virtual memory usage whereas it takes 438 seconds with a 1.8 GB peak virtual memory usage in a 2 node network configuration.

As discussed in this section, model checking eases the verification process. However for certain properties, especially for the safety properties, it can take time and high memory to verify the property. Although it seems that it is not feasible to use model checking for the verification of such properties, it is still feasible to reserve some time for verification since it is a one time process that is carried at the end of system design.

We give a detailed list of verification queries in Appendix C. The queries are given with the properties they verified. With the given list of queries the properties that are verified in Section 4.2.4 are verified via model checking.

6.2 Simulation of D^{3}RIP Framework

The protocol layers given in Section 5.6 are defined in a general way without the explicit definition of certain functions. This section defines a protocol family using the framework layers defined in Section 5.6 to make a simulation. The developed IL, CL and DP protocols are given in Sections 6.2.1, 6.2.2 and 6.2.3 respectively

6.2.1 Time-Slotted Interface Layer (TSIL)

The time-slotted interface layer (TSIL) protocol is a protocol member of the IL protocol family as described in Section 4.2.1.2. It was first described without dependability support in [10] and instantiates the decision variables $\texttt{vIL}_i^{\mathrm{d}}$ as well as the protocol functions f_{upd} and f_{my} for each node $i \in \mathcal{I}$. Here, $\texttt{vIL}_i^{\mathrm{d}}$ is equipped with three attributes: $\texttt{vIL}_i^{\mathrm{d}}.\texttt{cnt}$ cyclically counts the successive nRT slots, $\texttt{vIL}_i^{\mathrm{d}}.\texttt{cyc}$ is introduced such that the slot assignment cyclically repeats after $\texttt{vIL}_i^{\mathrm{d}}.\texttt{cyc}$ time slots, and the nRT slot set $\texttt{vIL}_i^{\mathrm{d}}.\texttt{nRTSet}$ of node i describes the time slots that can be used for nRT messages by IL_i. Naturally, it is required that $\texttt{vIL}_i^{\mathrm{d}}.\texttt{nRTSet} \cap \texttt{vIL}_j^{\mathrm{d}}.\texttt{nRTSet} = \emptyset$ for $i, j \in \mathcal{I}$, $i \neq j$. Furthermore, f_{upd} and f_{my} are given as follows.

$$\begin{aligned} f_{\mathrm{upd}}(\texttt{vIL}_i^{\mathrm{d}},\\ \texttt{RTIL}_i^{\mathrm{d}}).\texttt{cnt} \end{aligned} = \begin{cases} \texttt{vIL}_i^{\mathrm{d}}.\texttt{cnt} & \text{if } \texttt{RTIL}_i^{\mathrm{d}} = \textbf{true} \\ (\texttt{vIL}_i^{\mathrm{d}}.\texttt{cnt} + 1) & \\ \bmod \texttt{vIL}_i^{\mathrm{d}}.\texttt{cyc} & \text{otherwise} \end{cases}$$

$$\begin{aligned} f_{\mathrm{my}}(\texttt{vIL}_i^{\mathrm{d}},\\ \texttt{RTIL}_i^{\mathrm{d}}, b_2, i) \end{aligned} = \begin{cases} b_2 & \text{if } \texttt{RTIL}_i^{\mathrm{d}} = \text{true} \\ \text{true} & \text{if } \neg\texttt{RTIL}_i^{\mathrm{d}} \wedge \texttt{vIL}_i^{\mathrm{d}}.\texttt{cnt} \\ & \in \texttt{vIL}_i^{\mathrm{d}}.\texttt{nRTSet} \\ \text{false} & \text{otherwise.} \end{cases}$$

That is, the slot counter $\texttt{vIL}_i^{\mathrm{d}}.\texttt{cnt}$ is incremented by f_{upd} only in nRT slots. f_{my} is defined such that the ownership of RT slots is determined by the upper layer (variable b_2), while the ownership of nRT slots is locally decided by checking $\texttt{vIL}_i^{\mathrm{d}}.\texttt{nRTSet}$.

6.2.2 Urgency-Based Real-time Protocol (URT)

We refer to Section 4.2.1.3 for the general CL formulation. We present the *urgency-based real-time protocol* (URT) as an example of the CL protocol family. URT is designed to dynamically update information about the right to transmit, for each network device, in the form of *communication requests*. The decision variable is a *priority queue* $\mathsf{vCL}_i^{\mathrm{d}}.\mathrm{PQ}$, that holds requests in the form of a tuple (b, c, eT, dT), where b denotes a device, c is a channel, eT is an *eligibility time* and dT is a *deadline*, that is measured relative to the time instant, where the request is issued. Semantically, a request (b, c, eT, dT) states, that the device b can send the next message of the channel c after time eT, and must send the next message before dT. Accordingly, each message m transmitted on URT contains a set of requests m.par.req as its protocol parameter. Upon message reception, the requests are stored in $\mathsf{vCL}_i^{\mathrm{d}}.\mathrm{PQ}$, $i \in \mathcal{I}$, ordered by eligibility time and deadline, such that in each time slot, the device with the most urgent eligible request gets access to the medium. It has to be noted that the underlying assumption for this protocol is, that the urgency of requests is decided by the upper layer control application.

Formally, the update functions for URT are defined as follows. If $g_{\mathrm{upd}}(\mathsf{vCL}_i^{\mathrm{d}}, m.\mathrm{par}, t)$ is called and $\mathsf{RTCL}_i^{\mathrm{d}} = \text{true}$, the first request (i.e., the request that was eligible in the previous RT slot), is removed from $\mathtt{vCL}_i^{\mathrm{d}}$.PQ if m.par is not empty, i.e., a valid RT message has been received. Otherwise, the first request reenters $\mathtt{vCL}_i^{\mathrm{d}}$.PQ, since the required transmission did not happen, yet. Furthermore, all requests in *par*.req are inserted in $\mathtt{vCL}_i^{\mathrm{d}}$.PQ, after the current absolute time t is added to all relative times in each request. The remaining function definitions are.

$$
\begin{aligned}
g_{\mathrm{upd}}(\mathtt{PQ}_i^{\mathrm{d}}, m) &= \mathtt{PQ}_i^{\mathrm{d}}.\mathrm{Pop}(); \mathtt{PQ}_i^{\mathrm{d}}.\mathrm{Push}(m.\mathrm{ph}) \\
g_{\mathrm{RT}}(\mathtt{PQ}_i^{\mathrm{d}}, t) &= \begin{cases} \text{true} & \text{if } \mathtt{PQ}_i^{\mathrm{d}}.\mathrm{Top}().eT \leq t \\ \text{false} & \text{otherwise} \end{cases} \\
g_{\mathrm{my}}(\mathtt{PQ}_i^{\mathrm{d}}, i) &= \begin{cases} (\text{true}, a) & \text{if } \mathtt{PQ}_i^{\mathrm{d}}.\mathrm{Top}().b = i \\ & \wedge \mathtt{PQ}_i^{\mathrm{d}}.\mathrm{Top}().c = a \\ (\text{false}, 0) & \text{otherwise} \end{cases}
\end{aligned}
$$

That is, a slot is declared as RT slot, if the first request in the priority queue is eligible, and it belongs to the device i, if it is specified in that request.

6.2.3 Synchronization Based Dependability Protocol (SDEP)

We refer to Section 4.2.2 for the general definition of DP. We next realize the dependability plane for the previously described protocols TSIL and URT. The dependability header of the transmitted messages contains the vIL and vCL structures of the message transmitting node. To this end, it is necessary to instantiate the parameter vDL and the functions f_{oc}, f_{IL} and f_{CL}. Regarding f_{oc}, it is desired to deduce if a nRT slot is a transmission slot. According to the interface layer example TSIL in Section 6.2.1, this is the case if and only if the current nRT slot counter belongs to the nRT set $\mathtt{vIL}_i^{\mathrm{d}}.\mathtt{nRTSet}$ of some node $i \in \mathcal{I}$. Since this information is only available to node i but not to all other nodes, we use vDL as a set that contains the union of all nRT sets:

$$vDL = \bigcup_{i \in \mathcal{I}} \mathtt{vIL}_i^{\mathrm{d}}.\mathtt{nRTSet}. \tag{6.5}$$

Moreover, the nRT counter is available at each node i as $\mathtt{vIL}_i^{\mathrm{d}}.\mathtt{cnt}$. Hence, we use

$$f_{\mathrm{oc}}(\mathtt{vIL}_i^{\mathrm{d}}, vDL) = \begin{cases} \text{true} & \text{if } \mathtt{vIL}_i^{\mathrm{d}}.\mathrm{cnt} \in vDL \\ \text{false} & \text{otherwise} \end{cases} .$$

In order to check the consistency of the decision variables $\mathtt{vIL}_i^{\mathrm{d}}$ (TSIL) using f_{IL}, we compare the nRT slot counter of node i with the counter value received in the IL part vIL of the dependability header of any incoming nRT message. That is, we perform

$$f_{\mathrm{IL}}(\mathtt{vIL}_i^{\mathrm{d}}, vIL) = \begin{cases} \text{true} & \text{if } \mathtt{vIL}_i^{\mathrm{d}}.\mathtt{cnt} = vIL.\mathtt{cnt} \\ \text{false} & \text{otherwise} \end{cases} .$$

Finally, the correct protocol operation requires that the decision variables of the CL are identical in all nodes. That is, f_{CL} directly compares $\mathtt{vCL}_i^{\mathrm{d}}$ of node i and vCL from the dependability header of any incoming message.

$$f_{\mathrm{CL}}(\mathtt{vCL}_i^{\mathrm{d}}, vCL) = \begin{cases} \text{true} & \text{if } \mathtt{vCL}_i^{\mathrm{d}} = vCL \\ \text{false} & \text{otherwise} \end{cases}$$

6.2.4 Simulation Example

A three node simulation network is set up, by using the defined protocol family, in order to see the behavior of the developed dependability plane. The simulated network uses the shared medium fully utilized. Each of the nodes transmit a real-time message in its transmission slot. The simulation timing and the dynamic time slot assignments are given in Figure 6.2 ng and the dynamic time slot assignments are given in Figure 6.2

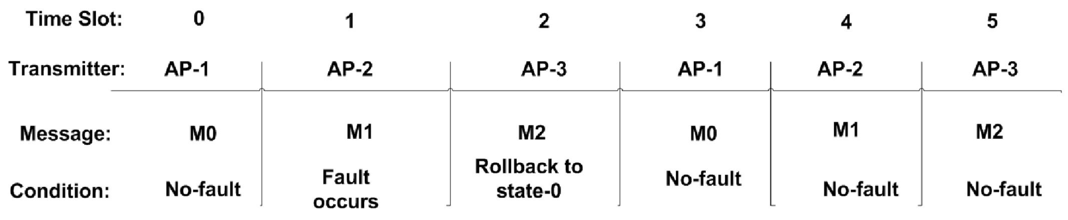


Figure 6.2: Timing of the Simulation Case Study

There are three applications, AP-1, AP-2 and AP-3, each running on a separate node. The simulation is started by AP-1 transmitting a real-time message $M0$. At the very beginning of the second time slot, AP-2 faces with an internal fault caused of corrupted vCL structure. When it transmits its message appending its vCL structure, the other healthy nodes receiving the message recognizes the fault condition. AP-3 having the third time slot transmits a rollback request to the network immediately. All the nodes observing the rollback request, rolls back to the state that is recorded at the first time-slot. From that point on the operation continues from the initial state by a RT message transmission from AP-1.

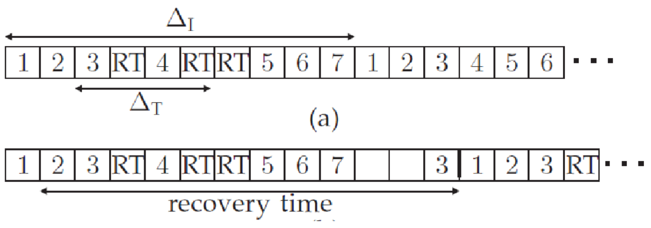
That is, the worst-case recovery time is indeed $dSlot \cdot \Delta_{\mathrm{F}}$ and should be bounded by 20 ms in RT applications according to [65]. An example for this computation is given in Section 6.2.5.

6.2.5 Worst-Case Recovery Delay Calculation Example

We refer to the application example in [10] in order to illustrate the rollback recovery. In this example, there is a set of 7 network nodes $\mathcal{I} = \{1, \cdots, 7\}$ that transmit both RT and nRT messages. The requests for RT messages have an eligibility time of 4 ms and it is known from the application that there are at most $Q_{\max} = 3$ requests in the priority queue $\mathtt{PQ}_i^{\mathrm{d}}$ of the CL at any time. Regarding the usage of TSIL as described in Section 6.2.1, we suggest a uniform distribution of nRT slots among the 7 nodes. That is, we use the following nRT sets for the nodes in $\mathcal{I}$: $\mathtt{vIL}_1^{\mathrm{d}}.\mathtt{nRTSet} = \{0\}$, $\mathtt{vIL}_2^{\mathrm{d}}.\mathtt{nRTSet} = \{1\}$, $\mathtt{vIL}_3^{\mathrm{d}}.\mathtt{nRTSet} = \{2\}$, $\mathtt{vIL}_4^{\mathrm{d}}.\mathtt{nRTSet} =$

{3}, $\texttt{vIL}_5^{\mathrm{d}}$.$\texttt{nRTSet}$ = {4}, $\texttt{vIL}_6^{\mathrm{d}}$.$\texttt{nRTSet}$ = {5}, $\texttt{vIL}_7^{\mathrm{d}}$.$\texttt{nRTSet}$ = {6}. The cycle parameter is $\texttt{vIL}_i^{\mathrm{d}}.\texttt{cnt} = 7$ for all $i \in \mathcal{I}$ and the DP parameter vDL evaluates to $vDL = \{0, \ldots, 6\}$.

In line with our experiments in [60], we assume a slot time of $dSlot = 250\,\mu$s. Considering that the eligibility time for requests is 4 ms (16 time slots) and there are at most 3 requests in $\texttt{PQ}_i^{\mathrm{d}}$, this implies that at most 3 out of 16 time slots are used as RT slots. A possible configuration is shown in Figure 6.3 (a).



Δ_I
1 2 3 RT 4 RT RT 5 6 7 1 2 3 4 5 6 ···
Δ_T
(a)
1 2 3 RT 4 RT RT 5 6 7 3 1 2 3 RT ···
recovery time

Figure 6.3: Protocol operation: (a) Fault-free slot sequence; (b) Slot sequence with rollback recovery.

Here, nRT slots are successively allocated to the different nodes, interrupted by RT slots that are used according to the requests in $\texttt{PQ}_i^{\mathrm{d}}$. From such allocation, the relevant delays Δ_I and Δ_T can be determined. Considering the inter-transmission delay Δ_I, there will always be 7 nRT slots and at most 3 RT slots between two transmission slots of any node. Hence, $\Delta_\mathrm{I} = 10$ as is shown in Figure 6.3 (a). Similarly, the transmission of three nodes is interrupted by at most 2 RT slots of the same nodes. The third RT slot or the next nRT slot must belong to a different node. Hence, $\Delta_\mathrm{T} = 4$ as is also shown in Figure 6.3 (a). Together, we obtain $\Delta_\mathrm{F} = 14$. That is, the worst-case fault recovery-time is $14 \cdot 250\,\mu\text{s} = 3.5\,\text{ms}$ which complies with the grace period of 20 ms for time-critical systems [65].

We finally describe a rollback scenario. Assume that node 1 encounters a software fault by a wrong update of its counter variable $\texttt{vIL}_1^{\mathrm{d}}.\texttt{cnt} = 2$ (instead of 1) after its first transmission slot. In that case, node 1 detects the inconsistency in its acceptance test in the next slot by applying $f_\mathrm{IL}(\texttt{vIL}_1^{\mathrm{d}}, vIL)$. Here, vIL comes in the protocol header in the nRT message that is sent by node 2. Since $\texttt{vIL}_1^{\mathrm{d}}.\texttt{cnt} = 2$ and $vIL.\texttt{cnt} = 1$, the acceptance test returns **false** and node 1 waits for its next transmission slot (after Δ_I time slots) to send a rollback message. However, due to the wrong counter variable, node 1 misses its next transmission slot and incorrectly transmits one slot later. The missed slot (with nRT counter value 0) is detected by all other nodes since they expect the transmission of an nRT messages ($0 \in vDL$). That is, the acceptance test of all nodes except for node 1 returns **false** and these nodes prepare a rollback message for their next transmission slot. Since node 1 incorrectly transmits in the transmission slot of node 2, a collision occurs and node 2 cannot successfully transmit the rollback message. Hence, node 3 in the next transmission slot provides a rollback messages and all nodes roll back to the confirmed correct state at the beginning of the operation. This scenario is illustrated in Figure 6.3 (b).

In this chapter, we have created a protocol family using the framework layers defined in Chapter 5. A network has been simulated by using the defined protocol family and an application layer automata. Simulation of the defined network has been carried out with two different network configurations and the worst-case recovery delay for our dependability design is calculated for an example. Finally we have verified the complete protocol family by using the UPPAAL tool-suite. Hence we have confirmed the applicability of TUConvert by a distributed real-time system example using simulation and formal verification of the converted layer models.

CHAPTER 7

CONCLUSIONS

In this thesis, we develop a generic dependability design methodology for the communication networks of distributed real-time systems. The proposed dependability plane is designed to work with broadcast capable and time slotted networks. Different from previous work, we focus on software fault-tolerance instead of hardware fault-tolerance. We first assess the potential fault scenarios and determine which faults can be detected by the distributed nodes. We further evaluate the maximum number of time slots that can pass until a fault is detected and fault resolution is initiated by some node.

Accordingly, we define a novel strategy for software fault tolerance that is based on synchronized checkpointing and rollback to a non-faulty protocol state. A main advantage of our strategy is the avoidance of additional communication messages by piggybacking dependability information on application messages that are transmitted by each node in a regular manner. The usage of protocol-specific information helps finding a close rollback state. Our dependability design is formalized by timed input/output automata (TIOA) models and its correctness is verified by formal proofs.

In addition, the formal verification of the design is carried out by the model checking software tool UPPAAL. To this end, we present an algorithmic approach for the conversion of TIOA models to UPPAAL timed automata (TA) models. First, basic conditions for TIOA models that are suitable for such conversion are established. Second, the algorithm TUConvert for the conversion of TIOA models to UPPAAL templates is developed. As a particular novel feature, this algorithm allows for the conversion of distributed real-time system models, that are composed of various TIOA models. In this case, separate UPPAAL 'TA models that are synchronized by broadcast channels are generated. Third, the practicability of the presented algorithm is demonstrated by its application to a distributed industrial communication protocol suite that was previously defined in the form of TIOA models. We implemented the tool TUConvert to automate the conversion.

We demonstrate the features of both the proposed dependability design and the TUConvert algorithm by applying them on a real-time communication protocol family. To this end, we develop the dependability plane by modeling it as TIOA. We then integrate the dependability plane model to the existing TIOA models of the protocol stack. The verification of the overall stack is carried out by converting the models to the UPPAAL TA models using TUConvert. It is shown that the desired properties of the protocol family can be successfully verified by using the verification engine of UPPAAL. Lastly we have built a simulation network in the UPPAAL environment and highlight the operational characteristics of the protocol family under an example fault scenario.

APPENDIX A

TUCONVERT INPUT FILE FORMAT

The input file format of TUConvert is designed according to the TIOA definition used in this report. The input file is designed in a tree format. The root of the file is formed by the *system*. *TIOA* forms the first branch. The TIOA name, parameters of a certain type, parameters defining variable types, and states of TIOA are written under the TIOA branch with *name, parameter, types* and *states* keywords respectively. The states of TIOA are defined by two branches. The *variables* branch is used for variable declerations whereas the *init* branch is used for initialiation.

The next sub-branch of TIOA is the *signatures*. Each automata signature is defined by *name, type, array, ID* and *size* fields. The array field of the signature is used to point whether the corresponding signature is a singleton or not. *ID* and *size* fields are not mandatory for singleton signatures. If the signature is used as an array the ID and size fields are used to define the array index and the array size of the signature. The signature parameters are defined in the *parameter* sub-branch of the signature.

The transitions of the TIOA are defined under the *transitions* branch. A transition is defined by the *name, type, parameter, array, ID* and *noTimeZones* fields. The ID field is not mandatory in case of a singleton signature. The noTimeZones field is used to define the number of disjuncts defined in the transition precondition. The transition behavior is defined by the *pre, paramassn* and *eff*. The pre field is defined for internal and output transitions and defines the transition preconditions whereas the paramassn field is defined for output transitions and is used to assign the action parameter. The eff field simply defines the effect field of the transition.

The final branch of TIOA is the *trajectories*. The evolve rate and stop conditions are defined under the *evolve rate* and *stop* sub-branches. After defining the automatons in the system, the input xml is ended with the system declerations that are defined under the *decleration* branch.

```
 <?xml version="1.0" encoding="utf-8"?>
<system>
<TIOA>
<name>UseOldInputA</name>
<parameter>int &maxIt</parameter>
```

```
<types/>
<states>
<variables>int maxoutA;
real nowA;
double nextA;
int infty;
</variables>
<init>maxoutA = maxIt;
nowA = 0;
nextA = 0;
infty = 1000;
</init>
</states>
<signatures>
<signature name="A" type="output" array="false" ID="" size="">
<parameter>int cnt</parameter>
</signature>
<signature name="B" type="input" array="false">
<parameter/>
</signature>
</signatures>
<transitions>
<transition name="A" type="output" parameter="int cnt"
array="false" ID="" noTimeZones='1'>
<pre>(maxoutA > 0) && (nowA == nextA )</pre>
<paramassn>cnt = maxoutA</paramassn>
<eff>maxoutA = maxoutA - 1;
nextA = infty;
</eff>
</transition>
<transition name="B" type="input" parameter="" array='false'>
<eff>if (nextA == infty)
{
nextA = nowA + 1;
}
</eff>
</transition>
</transitions>
<trajectories>
<stop>nowA == nextA</stop>
<evolve rate='1'>nowA</evolve>
</trajectories>
</TIOA>
```

```
<TIOA>
<name>UseOldInputB</name>
<parameter/>
<types/>
<states>
<variables>int maxoutB;
real nowB;
double nextB;
int infty;
</variables>
<init>maxoutB = 0;
nowB = 0;
nextB = 0;
infty = 1000;
</init>
</states>
<signatures>
<signature name="A" type="input" array="false">
<parameter>int cnt</parameter>
</signature>
<signature name="B" type="output" array="false">
<parameter/>
</signature>
</signatures>
<transitions>
<transition name="A" type="input" parameter="int cnt"
array='false' noTimeZones='1'>
<eff>maxoutB = cnt;
if (nextB == infty)
{
nextB = nowB + 1;
}
</eff>
</transition>
<transition name="B" type="output" parameter="" array="false" noTimeZones='1'>
<pre>(maxoutB > 0) && (nowB == nextB )</pre>
<paramassn/>
<eff>nextB = infty;</eff>
</transition>
</transitions>
<trajectories>
<stop>nowB == nextB</stop>
<evolve rate='1'>nowB</evolve>
```

```
</trajectories>
</TIOA>
<declaration>
int mxInteger = 100;
t1 = UseOldInputA(mxInteger);
t2 = UseOldInputB();
system t1,t2;
</declaration>
</system>
```

APPENDIX B

TUCONVERT OUTPUT FILE FORMAT

The output file format is compatible with the UPPAAL input file format.

```
<?xmlversion = ”1.0”encoding = ”utf − 8”? >
< nta >
< declaration >
chanA;
intcnt;
chanB;
< /declaration >
< template >
< name > UseOldInputA < /name >
< parameter > int&maxIt < /parameter >
< declaration > intmaxoutA;
clocknowA;
doublenextA;
intinfty;
voidInitAutomaton()
{
maxoutA = maxIt;
nowA = 0;
nextA = 0;
infty = 1000;
}
voidAfunc()
{
maxoutA = maxoutA − 1;
nextA = infty;
cnt = maxoutA; }
voidBfunc()
{
if(nextA == infty)
{
```

```
nextA = nowA + 1;
}
}
< /declaration >
< locationid = ”id0”x = ” − 250”y = ”0” >
< namex = ” − 250”y = ”20” > l_init < /name >
< committed/ >
< /location >
< locationid = ”id1”x = ”0”y = ”0” >
< namex = ”0”y = ”20” > l_idle < /name >
< labelkind = ”invariant”x = ”0”y = ”40” > nowA <= nextA < /label >
< /location >
< initref = ”id0”/ >
< transition >
< sourceref = ”id0”/ >
< targetref = ”id1”/ >
< labelkind = ”assignment”x = ” − 165”y = ”0” > InitAutomaton() < /label >
< /transition >
< transition >
< sourceref = ”id1”/ >
< targetref = ”id1”/ >
< labelkind = ”guard”x = ”167”y = ”125” > (maxoutA > 0)&&(nowA == nextA) <
/label >
< labelkind = ”assignment”x = ”167”y = ”145” > Afunc() < /label >
< labelkind = ”synchronisation”x = ”167”y = ”165” > A! < /label >
< nailx = ”0”y = ”500”/ >
< nailx = ”433”y = ”249”/ >
< /transition >
< transition >
< sourceref = ”id1”/ >
< targetref = ”id1”/ >
< labelkind = ”synchronisation”x = ”50”y = ” − 250” > B? < /label >
< labelkind = ”assignment”x = ”50”y = ” − 230” > Bfunc() < /label >
< nailx = ”433”y = ” − 250”/ >
< nailx = ”0”y = ” − 500”/ >
< /transition >
< /template >
< template >
< name > UseOldInputB < /name >
< parameter/ >
< declaration > intmaxoutB;
clocknowB;
doublenextB;
```

```
intinfty;
voidInitAutomaton()
{
maxoutB = 0;
nowB = 0;
nextB = 0;
infty = 1000;
}
voidAfunc()
{
maxoutB = cnt;
if(nextB == infty)
{
nextB = nowB + 1;
}
}
voidBfunc()
{
nextB = infty; }
< /declaration >
< locationid = ”id0”x = ” − 250”y = ”0” >
< namex = ” − 250”y = ”20” > l_init < /name >
< committed/ >
< /location >
< locationid = ”id1”x = ”0”y = ”0” >
< namex = ”0”y = ”20” > l_idle < /name >
< labelkind = ”invariant”x = ”0”y = ”40” > nowB <= nextB < /label >
< /location >
< initref = ”id0”/ >
< transition >
< sourceref = ”id0”/ >
< targetref = ”id1”/ >
< labelkind = ”assignment”x = ” − 165”y = ”0” > InitAutomaton() < /label >
< /transition >
< transition >
< sourceref = ”id1”/ >
< targetref = ”id1”/ >
< labelkind = ”synchronisation”x = ”167”y = ”125” > A? < /label >
< labelkind = ”assignment”x = ”167”y = ”145” > Afunc() < /label >
< nailx = ”0”y = ”500”/ >
< nailx = ”433”y = ”249”/ >
< /transition >
< transition >
< sourceref = ”id1”/ >
< targetref = ”id1”/ >
< labelkind = ”guard”x = ”50”y = ” − 250” > (maxoutB > 0)&&(nowB == nextB) <
/label >
< labelkind = ”assignment”x = ”50”y = ” − 230” > Bfunc() < /label >
< labelkind = ”synchronisation”x = ”50”y = ” − 210” > B! < /label >
< nailx = ”433”y = ” − 250”/ >
< nailx = ”0”y = ” − 500”/ >
< /transition >
< /template >
< system >
intmxInteger = 100;
t1 = UseOldInputA(mxInteger);
t2 = UseOldInputB();
systemt1, t2;
< /system >
< /nta >
```

APPENDIX C

TCTL QUERIES FOR D^{3}RIP VERIFICATION

- $A\Box$ *not deadlock*
  - Safety property stating the deadlock freedom of the system that is composed of SM, IL, CL and DL
- $A\Box\ (DL1.stNo\ ==\ 3)\ \ \&\ \&\ \ (DL1.nowDL\ ==\ t0)\ imply\ IL.noStt\ ==\ 26$
  - Safety property stating that within a finite time interval, it is not possible for IL to make infinite number of state transitions. (i.e there is no locally Zeno behavior.) So IL is progressive.

    Here *noStt* is a counter that is defined to count the number of state transitions. In the implemented network configuration, IL should take 26 state transitions when DL triggers the rollback action.
- $E\Diamond$ *IL.l_idle*
  - Reachability property stating that *l_idle* state of IL is reachable.
- $E\Diamond$ *CL.l_idle*
  - Reachability property stating that *l_idle* state of CL is reachable.
- $E\Diamond$ *DL.l_idle*
  - Reachability property stating that *l_idle* state of DL is reachable.
- $A\Box\ (IL1.bReqIL\ ==\ true)\ \ \&\ \&\ \ (IL1.nowIL\ ==\ t1)\ imply\ IL2.nowIL\ ==\ t1$
  - Safety property stating that the protocol layers located on the distributed nodes operates synchronously for a 2 node network.
- $A\Box\ (IL1.bReqIL\ ==\ true)\ \ \&\ \&\ \ (IL1.nowIL\ ==\ t1)\ imply\ IL3.nowIL\ ==\ t1$
  - Safety property stating that the protocol layers located on the distributed nodes operates synchronously for a 3 node network.
- $A\Box\ (CL1.vCL.iCnt\ ==\ CL2.vCL.iCnt)\ \ \&\ \&\ \ (CL1.sendCL\ ==\ CL2.sendCL)\ imply\ (CL1.vCL\ ==\ CL2.vCL)$

  - Safety property stating that CL located on the distributed nodes on a 2 node network has consistent state variables hence D^{3}RIP is suitable for synchronized checkpointing

- $A\Box\,(CL1.vCL.iCnt\ ==\ CL3.vCL.iCnt)\ \ \&\ \&\ \ (CL1.sendCL\ ==\ CL3.sendCL)\, imply\ (CL1.vCL\ ==\ CL3.vCL)$

  - Safety property stating that CL located on the distributed nodes on a 3 node network has consistent state variables hence D^{3}RIP is suitable for synchronized checkpointing

- $A\Box\,(IL1.nowIL\ ==\ IL2.nowIL)\ \ \&\ \&\ \ (IL1.vIL.iCnt\ ==\ IL2.vIL.iCnt)\, imply\ IL1.vIL\ ==\ IL2.vIL$

  - Safety property stating that IL located on the distributed nodes on a 2 node network has consistent state variables hence D^{3}RIP is suitable for synchronized checkpointing

- $A\Box\,(IL1.nowIL\ ==\ IL3.nowIL)\ \ \&\ \&\ \ (IL1.vIL.iCnt\ ==\ IL3.vIL.iCnt)\, imply\ IL1.vIL\ ==\ IL3.vIL$

  - Safety property stating that IL located on the distributed nodes on a 3 node network has consistent state variables hence D^{3}RIP is suitable for synchronized checkpointing

- $A\Box\,(DL1.stNo\ ==\ 3)\ \ \&\ \&\ \ (DL1.nowDL\ ==\ t0)\, imply\, GmessILSM.par.ATRes == false$

  - Safety property stating that DL of node 1 trigs the rollback process if the acceptance test result received from the message transmitting node is false.

REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[2] D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager, *The Theory of Timed IO Automata, Second Edition*. Morgan and Claypool Publishers, 2010.

[3] T. Sauter and A. Treytl, “Communication systems as an integral part of distributed automation systems,” in *Distributed Manufacturing*, H. Kühnle, Ed. Springer London, 2010, pp. 93–111.

[4] T. Sauter, S. Soucek, W. Kastner, and D. Dietrich, “The evolution of factory and building automation,” *Industrial Electronics Magazine, IEEE*, vol. 5, no. 3, pp. 35–48, 2011.

[5] P. Gaj, J. Jasperneite, and M. Felser, “Computer communication within industrial distributed environment — a survey,” *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 182–189, 2013.

[6] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, “Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems,” *Real-Time Systems Symposium, IEEE International*, vol. 0, p. 166, 2003.

[7] P. P. Kim G. Larsen and W. Yi, “Uppaal in a nutshell,” 1994.

[8] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, “A survey of rollback-recovery protocols in message-passing systems,” *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[9] D. E. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, “The theory of timed I/O automata,” MIT Laboratory for Computer Science, Cambridge, MA, Tech. Rep. MIT-LCS-TR-917, 2003.

[10] K. Schmidt and E. Schmidt, “Distributed real-time protocols for industrial control systems: Framework and examples,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 10, pp. 1856–1866, 2012.

[11] G. Buja and R. Menis, “Dependability and functional safety: Applications in industrial electronics systems,” *Industrial Electronics Magazine, IEEE*, vol. 6, no. 3, pp. 4–12, 2012.

[12] Y. Huang and C. M. R. Kintala, “Software implemented fault tolerance technologies and experience,” in *International Symposium on Fault-Tolerant Computing*, 1993, pp. 2–9.

[13] R. Koo and S. Toueg, “Checkpointing and rollback-recovery for distributed systems,” *Software Engineering, IEEE Transactions on*, vol. SE-13, no. 1, pp. 23–31, 1987.

[14] G.-M. Chiu and C.-R. Young, "Efficient rollback-recovery technique in distributed computing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, no. 6, pp. 565–577, 1996.

[15] P. Ramanathan and K. G. Shin, "Use of common time base for checkpointing and rollback recovery in a distributed system," *IEEE Trans. Softw. Eng.*, vol. 19, no. 6, pp. 571–583, Jun. 1993.

[16] J. Tsai, "Flexible symmetrical global-snapshot algorithms for large-scale distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 3, pp. 493–505, 2013.

[17] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Operating Systems, Proceedings of an International Symposium.* London, UK, UK: Springer-Verlag, 1974, pp. 171–187.

[18] B. Randell, "System structure for software fault tolerance," *Software Engineering, IEEE Transactions on*, vol. 10, no. 6, pp. 437–449, Apr. 1975.

[19] J.-M. Helary, A. Mostefaoui, and M. Raynal, "Communication-induced determination of consistent snapshots," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 9, pp. 865–877, 1999.

[20] A. C. Simon, S. E. P. Hernandez, J. R. P. Cruz, and P. G.-G. K. Drira, "A scalable communication-induced checkpointing algorithm for distributed systems," *IEICE Transactions on Information and Systems*, vol. 96, no. 4, pp. 886–896, 2013.

[21] Z. Tong, R. Kain, and W. Tsai, "Rollback recovery in distributed systems using loosely synchronized clocks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 2, pp. 246–251, 1992.

[22] S. Neogy, A. Sinha, and P. Das, "Distributed checkpointing using synchronized clocks," in *International Computer Software and Applications Conference*, 2002, pp. 199–204.

[23] S. J. Garland, D. Kaynar, N. A. Lynch, J. A. Tauber, and M. Vaziri, "Tioa tutorial," 2005.

[24] C. Constant, T. Jeron, H. Marchand, and V. Rusu, "Integrating formal verification and conformance testing for reactive systems," *Software Engineering, IEEE Transactions on*, vol. 33, no. 8, pp. 558 –574, aug. 2007.

[25] Y. Zhu and T. Marshall, "Design verification using formal techniques," in *ASIC, 2001. Proceedings. 4th International Conference on*, 2001, pp. 21 –28.

[26] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, jun 1990, pp. 414 –425.

[27] A. Ayoub, A. M. Wahba, A. M. Salem, and M. A. Sheirah, "Tctl-based verification of industrial processes." in *FDL.* ECSI, 2003, pp. 456–468. [Online]. Available: http://dblp.uni-trier.de/db/conf/fdl/fdl2003.htmlAyoubWSS03

[28] F. Miyawaki, K. Masamune, S. Suzuki, K. Yoshimitsu, and J. Vain, "Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery," *Industrial Electronics, IEEE Transactions on*, vol. 52, no. 5, pp. 1227–1235, oct. 2005.

[29] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems," *Software Engineering, IEEE Transactions on*, vol. 30, no. 9, pp. 613–629, sept. 2004.

[30] H. Bel Mokadem, B. Berard, V. Gourcuff, O. De Smet, and J. Roussel, "Verification of a timed multitask system with uppaal," *Automation Science and Engineering, IEEE Transactions on*, vol. 7, no. 4, pp. 921 –932, oct. 2010.

[31] G. Behrmann, R. David, and K. G. Larsen, "A tutorial on uppaal." Springer, 2004, pp. 200–236.

[32] R. Alur and D. L. Dill, "Automata for modeling real-time systems," in *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 322–335. [Online]. Available: http://dl.acm.org/citation.cfm?id=90397.90438

[33] M. Mikucionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard, "Schedulability analysis using UPPAAL: Herschel-planck case study," in *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ser. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 175–190. [Online]. Available: http://portal.acm.org/citation.cfm?id=1939345.1939369

[34] S. Li, S. Balaguer, A. David, K. G. Larsen, B. Nielsen, and S. Pusinskas, "Scenario-based verification of real-time systems using UPPAAL," *Form. Methods Syst. Des.*, vol. 37, no. 2–3, pp. 200–264, dec. 2010.

[35] F. Wang, G.-D. Huang, and F. Yu, "Tctl inevitability analysis of dense-time systems: From theory to engineering," *Software Engineering, IEEE Transactions on*, vol. 32, no. 7, pp. 510 –526, july 2006.

[36] W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving," in *Proc. of the 7th International Conference on Formal Description Techniques*, 1994, pp. 223–238.

[37] J. Berendsen and F. Vaandrager, "Compositional abstraction in real-time model checking," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, F. Cassez and C. Jard, Eds. Springer Berlin Heidelberg, 2008, vol. 5215, pp. 233–249. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85778-5_17

[38] "Uppaal web help," [Available Online]: http://www.it.uu.se/research/group/darts/uppaal/help.php?file=WebHelp.

[39] J. Moyne and D. Tilbury, "The emergence of industrial control networks for manufacturing control, diagnostics, and safety data," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 29–47, Jan. 2007.

[40] P. Neumann, "Communication in industrial automation - what is going on?" *Control Engineering Practice*, vol. 15, pp. 1332–1347, 2007.

[41] L. Seno, F. Tramarin, and S. Vitturi, "Performance of industrial communication systems: Real application contexts," *Industrial Electronics Magazine, IEEE*, vol. 6, no. 2, pp. 27–37, 2012.

[42] A. Willig, K. Matheus, and A. Wolisz, "Wireless technology in industrial networks," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1130–1151, June 2005.

[43] G. Prytz, "Redundancy in industrial Ethernet networks," in *Factory Communication Systems, IEEE International Workshop on*, 2006, pp. 380–385.

[44] A. Giorgetti, F. Cugini, F. Paolucci, L. Valcarenghi, A. Pistone, and P. Castoldi, "Performance analysis of media redundancy protocol (MRP)," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 218–227, 2013.

[45] K. Hansen, "Redundancy Ethernet in industrial automation," in *Emerging Technologies and Factory Automation, IEEE Conference on*, vol. 2, 2005, pp. 941–947.

[46] H. Kirrmann and D. Dzung, "Selecting a standard redundancy method for highly available industrial networks," in *Factory Communication Systems, IEEE International Workshop on*, 2006, pp. 386–390.

[47] C.-H. Chen and N.-F. Huang, "Lib: A last-in-backup based fast recovery scheme for ring-based industrial networks," *Communications Letters, IEEE*, vol. 15, no. 6, pp. 680–682, 2011.

[48] M. Huynh, S. Goose, P. Mohapatra, and R. Liao, "RRR: Rapid ring recovery submillisecond decentralized recovery for Ethernet ring," *Computers, IEEE Transactions on*, vol. 60, no. 11, pp. 1561–1570, 2011.

[49] "Real-time Ethernet: EPL (Ethernet powerlink): Proposal for a publicly available specification for real-time Ethernet," Doc. IEC 65C/356a/NP, 2004.

[50] P. Pedreiras, P. Gai, L. Almeida, and G. Buttazzo, "FTT-Ethernet: a flexible real-time communication protocol that supports dynamic qos management on ethernet-based systems," *Industrial Informatics, IEEE Transactions on*, vol. 1, no. 3, pp. 162–172, 2005.

[51] G. Prytz and J. Skaalvik, "Redundant and synchronized EtherCAT network," in *Industrial Embedded Systems, International Symposium on*, 2010, pp. 201–204.

[52] A. Xu, L. Jiang, and H. Yu, "Research of fault-tolerance technique for high availability industrial Ethernet," in *Information and Automation, International Conference on*, 2009, pp. 301–305.

[53] "Real-time Ethernet: SERCOS III: Proposal for a publicly available specification for real-time Ethernet," Doc. IEC 65C/358/NP, 2004.

[54] "Real-time Ethernet: TCnet (Time-Critical Control Network): Proposal for a publicly available specification for real-time Ethernet," Doc. IEC 65C/353/NP, 2004.

[55] J. Lisner, "Efficiency of dynamic arbitration in TDMA protocols," in *Dependable Computing - EDCC 5*, ser. Lecture Notes in Computer Science, M. Cin, M. Kaâniche, and A. Pataricza, Eds. Springer Berlin Heidelberg, 2005, vol. 3463, pp. 91–102.

[56] (2002, Nov.) IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. [Online]. Available: http://ieee1588.nist.gov

[57] W. Hummer, C. Inzinger, P. Leitner, B. Satzger, and S. Dustdar, “Deriving a unified fault taxonomy for event-based systems,” in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2012, pp. 167–178.

[58] P.-L. Lai, J. J. M. Tan, C.-P. Chang, and L.-H. Hsu, “Conditional diagnosability measures for large multiprocessor systems.” *IEEE Trans. Computers*, vol. 54, no. 2, pp. 165–175, 2005. [Online]. Available: http://dblp.uni-trier.de/db/journals/tc/tc54.html#LaiTCH05

[59] E. P. Duarte, E. P. Duarte, T. Nanya, and T. Nanya, “A hierarchical adaptive distributed system-level diagnosis algorithm a hierarchical adaptive distributed system-level diagnosis algorithm,” *Transactions on Computers*, vol. 47, no. 1, pp. 34–45, 1998. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=656078

[60] A. Kaya, E. G. Schmidt, K. W. Schmidt, and T. Moor, “Dynamic distributed real-time industrial Ethernet Protocol (D^{2}RIP): Architecture, implementation and experimental evaluation,” *Industrial Informatics, IEEE Transactions on*, 2013.

[61] O. B. Sezer, “Implementation and evaluation of the dependability plane for the dynamic distributed dependable real time industrial protocol (d^3rip),” Master’s thesis, Middle East Technical University, Department of Electrical and Electronics Engineering, 2013.

[62] C. M. Robson, “Timed input/output automata and uppaal,” Master’s thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[63] Y. Kartal, “Tuconvert tool [online],” 2014. [Online]. Available: http://www.eee.metu.edu.tr/~eguran/D3RIP/TUConvert_pkg.exe

[64] S. Mondal and S. Sural, “Security analysis of temporal-rbac using timed automata,” in *Information Assurance and Security, 2008. ISIAS ’08. Fourth International Conference on*, sept. 2008, pp. 37 –40.

[65] M. Felser, “Media redundancy for profinet io,” in *Factory Communication Systems, 2008. WFCS 2008. IEEE International Workshop on*, 2008, pp. 325–330.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Kartal, Yusuf Bora
Nationality: Turkish (TC)
Date and Place of Birth: 13.04.1983, Ankara
Marital Status: Married
Phone: +905052293779

EDUCATION

Degree	Institution	Year of Graduation
M.S.	METU/Electrical and Electronics Engineering	2007
B.S.	METU/Electrical and Electronics Engineering	2005

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
2010 October-	ASELSAN Inc.	Radar Systems Design Engineer
2004 November-2010 October	ASELSAN Inc.	Software Engineer

FOREIGN LANGUAGES

Advanced English, Good German

PUBLICATIONS

International Conference Publications

- Y.B. Kartal, E. G. Schmidt, K. W. Schmidt, "Modeling and Formal Verification of Distributed Real-Time Systems Using Timed Input/Output Automata and UPPAAL ", 2014 (submitted for review)

- Y.B. Kartal, E. G. Schmidt, K. W. Schmidt, “Dependability Design for a Distributed Industrial Real-Time Protocol Family”, 2014 (under preperation)
- Y.B. Kartal, E. G. Schmidt, K. W. Schmidt, “The Verification of a Novel Framework for Real-Time Shared Medium Communication Network Protocols”, 2012
- Ç. Turan, A. Dökmen, S. Akdağ, Y.B. Kartal, “Gömülü Yazılım Geliştirme Pratikleri”, 2009
- Y.B. Kartal, E. G. Schmidt, “ İlgiye Odaklı Programlamanın Gerçek Zamanlı Gömülü Sistemler Üzerinde Bir Değerlendirmesi”, 2007
- Y.B. Kartal, E. G. Schmidt, “An Evaluation of Aspect Oriented Programming for Embedded Real-Time Systems”, 2007

HOBBIES

Swimming, Reading, Music