UNIVERSITAT DE LES ILLES BALEARS

DEPARTAMENT DE CIÈNCIES MATEMÀTIQUES I INFORMÀTICA
PROGRAMA DE DOCTORAT D'INFORMÀTICA

PH.D. THESIS

Design and Formal Verification of a Fault-tolerant Clock Synchronization Subsystem for the Controller Area Network

Guillermo Rodríguez-Navas González

SUPERVISOR
Julián Proenza Arenas

Faig constar que aquesta memòria ha estat realitzada, sota la direcció de Julián Proenza Arenas, per Guillermo Rodríguez-Navas González i que constitueix la seva tesi doctoral.

Palma, juny de 2010

Signat: Guillermo Rodríguez-Navas González
Estudiant de doctorat

Signat: Julián Proenza Arenas
Director de la tesi
Professor Titular d'Universitat
Departament de Ciències Matemàtiques i Informàtica
Universitat de les Illes Balears

A mis padres.

ॐ सह नाववतु |
सह नौ भुनक्तु |
सह वीर्यं करवावहै |
तेजस्विनावधीतमस्तु मा विद्विषावहै ||
ॐ शान्तिः शान्तिः शान्तिः ||

Auṃ saha nāvavatu
saha nau bhunaktu
saha vīryaṃ karavāvahai
tejasvināvadhītamastu mā vidviṣāvahai
Auṃ śāntiḥ śāntiḥ śāntiḥ

Aum. May He protect both of us
May He guide both of us
May we work together with vigor
May our study be brilliant, may there be no dislike between us
Aum. Peace, Peace, Peace.

Shanti Mantra, from the *Upanishads*.

Resum

El bus de camp Controller Area Network és una tecnologia amplament utilitzada en l'actualitat per al disseny de sistemes encastats distribuïts. Malgrat haver-se proposat inicialment per comunicacions dins vehicles, CAN s'ha convertit en un estàndard de facto per tot tipus de comunicacions encastades; la qual cosa ha reduït el seu cost significativament. Aquest cost assequible, juntament amb la seva bona resposta en temps real i les potents funcionalitats que incorpora per control d'errors, han generat molt d'interés en l'aplicabilitat de CAN per sistemes més exigents, com ara el sistemes distribuïts crítics.

No obstant això, i a pesar del seu indiscutible èxit en tants de camps d'aplicació, l'adopció de CAN per desenvolupar sistemes distribuïts crítics és encara una qüestió controvertida. La principal raó d'aquesta controvèrsia és que CAN pateix algunes limitacions relacionades amb la seva garantia de funcionament. Alguns investigadors ja han treballat sobre aquest problema i vàries tècniques per tal de resoldre cada una d'aquestes limitacions ja han estat proposades. Per tot això, malgrat algunes qüestions romanen obertes, sembla que aquest esforç combinat farà de CAN una solució vàlida fins i tot per sistemes crítics.

La motivació de la nostra recerca és ajudar en aquest intent de millorar les propietas relacionades amb garantia de funcionament de CAN. La nostra aproximació consisteix a centrar-nos en una de les limitacions que, en la nostra opinió, mereix més atenció: la falta d'un servei de sincronització de rellotge. Els autors defensam que la sincronització de rellotge és un requisit imprescindible per aconseguir garantia de funcionament sobre CAN, i que cal proporcionar aquest servei de tal manera que sigui garantida una precisió de l'ordre de microsegons inclús en presència de fallades de canal o dels nodes, a un cost raonable.

En aquesta dissertació describim una nova solució per sincronització de rellotge sobre CAN que ha estat dissenyada per satisfer aquests requisits. Aquesta solució, la qual hem anomenat Orthogonal Clock Subsystem for CAN (OCS-CAN), persegueix la consecució de quatre atributs específics: alta precisió, baixa sobrecàrrega, tolerància a fallades i baix cost.

OCS-CAN està format per un conjunt de dispositius hardware independents, que anomenam clock units, els quals s'adjunten als nodes del sistema. Cada clock unit disposa del seu propi controlador CAN per tal d'intercanviar missatges de sincronització amb les altres clock units. Per tant, el conjunt de clock units constitueix un subsistema independent que s'encarrega de proporcionar el servei de sincronització de rellotge sense importar l'aplicació concreta executada pels nodes. Diem que aquest subsistema és ortogonal perquè ha estat concebut com un afegit que pot ser incorporat a un altre sistema sense haver de reemplaçar cap component hardware ni haver de fer canvis als programes executats pels nodes.

Les clock units estan equipades amb una circuiteria específica, com ara un mecanisme per posar segells temporals a cada trama enviada o rebuda, que millora la precisió de la sincronització de rellotge i fan el rang the microsegons viable. Per tal de reduir les sobrecàrregues computacional i de comunicacions, l'algoritme de sincronització de rellotge implementat per OCS-CAN està basat en un esquema mestre/esclau. A més, donat que l'existència d'un únic mestre suposaria un punt singular d'avaria per al sistema, aquest algoritme empra replicació del meste i inclou uns mecanismes per detectar i reemplaçar un master avariat.

El desenvolupament i testeig d'un primer prototipus de OCS-CAN també és tractat en aquesta dissertació. L'objectiu del nostre prototipat és comprovar que l'arquitectura suggerida pot ser implementada amb un cost relativament baix. Al nostre cas, el prototipus s'ha construït amb un dispositiu de lògica programable tipus FPGA de gamma mitja.

Conjuntament amb la descripció de l'arquitectura d'OCS-CAN, una part significativa d'aquesta dissertació tracta sobre l'avaluació formal dels mecanismes de tolerància a fallades d'OCS-CAN. Aquesta avaluació s'ha realitzat de forma analítica i també mitjançant model checking. L'avaluació analítica es basa en aproximar els rellotges de les clock units amb funcions lineals a trossos, amb punts de discontinuïtat que coincideixen amb les accions de sincronització. El model checking s'ha acomplit amb el model checker UPPAAL, que és una eina basada en la teoria d'autòmats temporitzats i que és emprada habitualment per verificar formalment sistemes de temps real.

Els resultats obtinguts, tant a l'anàlisi com al model checking, demostren que els mecanismes de tolerancia a fallades d'OCS-CAN funcionen correctament i que és possible determinar la precisió garantida per diferents hipòtesis de fallades. Però, el que podria ser més interessant és que aquesta avaluació també proporciona informació valuosa sobre la relació entre les fallades potencials del sistema i la precisió garantida. En concret, mostra que OCS-CAN exhibeix una propietat coneguda com a degradació elegant o útil, ja que la precisió empitjora gradualment a mesura que els escenaris de fallades es tornen més complexes i greus. En la nostra opinió, el fet de proporcionar no només una solució adequada per sincronització de rellotge sobre CAN, sinó també donar els mitjans apropiats per avaluar el seu rendiment (en aquest cas, la precisió) és el camí que cal seguir per tal de desenvolupar una arquitectura sobre CAN adequada per a aplicacions amb garantia de funcionament.

Una contribució important d'aquesta dissertació, i que va més enllà de l'objectiu que ens havíem fixat de dissenyar un servei de sincronització de rellotge, és el detallat estudi d'una sèrie de patrons de modelització que permeten l'especificació realista, mitjançant autòmats temporitzats, de sistemes distribuïts amb rellotges. Dins d'aquest estudi, també presentam una nova tècnica de modelització, anomenada punters a rellotges, desenvolupada en principi per a la verificació formal d'OCS-CAN però que permet l'espeficicació d'altres algoritmes de sincronització de rellotge afectats per inconsistències transitòries. Aquesta tècnica estén l'aplicabilitat del formalisme del autòmats temporitzats i podria, a més a més, ser útil en el context general de la verificació formal de sistemes híbrids.

Abstract

The Controller Area Network (CAN) is a fieldbus technology that is nowadays extensively used for the design of distributed embedded systems. Although initially intended for in-vehicle communications, CAN has become a de facto standard for embedded communications, what has importantly decreased the cost of this technology. Said cost effectiveness, together with the good real-time performance as well as the potent error control capabilities it provides, has risen some significant interest on the applicability of CAN to more demanding systems, such as critical embedded systems.

Nevertheless, despite its uncontested success in so many application domains, the adoption of CAN for developing critical distributed embedded systems is still controversial. The main reason for this controversy is that CAN exhibits many limitations with respect to dependability. Some researchers have already addressed this matter, and several techniques have been proposed for overcoming each one of the reported limitations. Thus, although there are still open issues, it seems that this combined research effort will make CAN a valid solution even for critical systems.

The motivation of our research is to help in this attempt to improve the dependability properties of CAN. We do it by bringing into focus one of the limitations that, in our opinion, merits more attention: the lack of a clock synchronization service. We claim that clock synchronization is a requirement that is implicit in many of the techniques that allow the improvement of dependability over CAN, and that this service must be provided in a way such that a precision in the order of a few microseconds is guaranteed even in the presence of either channel or node faults, at a reasonable cost.

In this dissertation we describe a novel solution for clock synchronization over CAN that specifically addresses these requirements. This solution, which we have named the Orthogonal Clock Subsystem for CAN (OCS-CAN), pursues the fulfillment of four specific attributes: high precision, low overhead, fault tolerance and low cost.

OCS-CAN is made up of a set of independent hardware devices, called clock units, which are attached to the nodes of the system. Each clock unit is provided with a CAN controller for exchanging synchronization messages with the other clock units. Therefore, the set of clock units constitutes an independent subsystem that is aimed at supplying the clock synchronization service regardless of the application executed by the nodes. We say that this subsystem is orthogonal because it has been conceived as an add-on feature, which can be incorporated without the replacement of any hardware component nor any change of the software executed by the nodes.

The clock units are equipped with a number of hardware mechanisms, such as a mechanism for timestamping each frame transmitted or received, which improve the precision of the clock synchronization and make the range of microseconds reachable. In order to reduce the computation and communication overheads, the clock synchronization algorithm implemented by OCS-CAN follows

a master/slave scheme. Furthermore, and given that having only one master would represent a single point of failure of the system, this algorithm uses master replication and includes some mechanisms for detecting and replacing a faulty master.

The development and testing of a first prototype of OCS-CAN is also addressed in this dissertation. The aim of said prototype is to show that the suggested architecture can be implemented with low cost hardware. In our case, the prototype is built on a medium range FPGA.

Besides the description of the OCS-CAN architecture, a significant part of this dissertation deals with the formal assessment of the fault tolerance mechanisms of OCS-CAN. This assessment is performed both analytically and by means of model checking. The analytical assessment is addressed by approximating the clocks of the clock units as piecewise linear functions, with points of discontinuity that coincide with the synchronization actions. Model checking is performed with the UPPAAL model checker, which is a tool based on the theory of timed automata that is commonly used for formally verifying real-time systems.

The results obtained, both analytically and by model checking, prove that the fault tolerance mechanisms of OCS-CAN work as intended and that it is possible to determine the precision guaranteed under the considered fault assumptions. But, perhaps more importantly, this assessment also provides very valuable information about the relationship that exists between the potential faults of the system and the achievable precision. It shows that OCS-CAN exhibits graceful degradation of the precision, since it can be observed that the guaranteed precision worsens gradually as the fault scenarios become more complex and severe. In our opinion, providing not only a suitable solution for clock synchronization over CAN, but also the proper means to assess its performance (i.e. the precision) against different fault conditions, is the way to follow in order to develop an adequate architecture for supporting dependable applications over CAN.

An important contribution of this dissertation, which goes beyond our initial goal of just designing a clock synchronization service, is the thorough study of a series of modeling patterns for the realistic specification, by means of timed automata, of distributed systems with computer clocks. Within this study, we also present a novel modeling technique, named clock pointers, which we have developed in principle for the formal verification of OCS-CAN but that has a wider applicability, since it allows the specification of any clock synchronization algorithm affected by transient inconsistencies. This technique then extends the applicability of the timed automata formalism and might indeed be useful in the general context of the formal verification of hybrid systems.

Acknowledgments

En estas líneas me gustaría expresar mi agradecimiento a todas las personas que de una manera u otra, voluntaria o involuntariamente, me han ayudado a hacer posible este trabajo. Han sido muchos años de trabajo y ha sido mucha la gente que me ha apoyado, así que no espero poder recordarlos a todos. Por ello, antes que nada, un enorme "Gracias a todos".

Como no puede ser de otra manera, en primer lugar quiero agradecer a mi director de tesis, Julián Proenza, su incansable e incondicional apoyo durante todo este tiempo. La posibilidad de comenzar una carrera de investigador surgió hace muchos años en una conversación casual entre nosotros. Por aquel entonces yo no podía concebir siquiera el fascinante mundo que se escondía "al otro lado del espejo", es decir, de los *papers*... Siempre le estaré agradecido por darme la posibilidad de trabajar con él, así como por su paciencia y dedicación en mis primeros años de investigación, en los cuales me enseñó prácticamente todo sobre el oficio y la ética del investigador.

También quiero dar las gracias a todos mis compañeros del grupo de *Sistemes, Robòtica i Visió* (SRV). A Gabriel Oliver, el director del grupo, por poner a nuestra disposición todos los medios materiales a su alcance y por su continuo esfuerzo en tareas burocráticas, sin perder el buen humor. A Alberto Ortiz y a Yolanda González, por estar siempre disponibles para ayudar y por ser un ejemplo a seguir. A mis compañeros desde casi el primer día, Javier Antich, Toni Burguera y José Guerrero (*cited in no particular order*), por las muchas conversaciones y risas compartidas, por darme apoyo moral en múltiples ocasiones, y por ser excelentes compañeros, con una capacidad de trabajo y un compromiso que resultan realmente inspiradores. A los compañeros que han llegado un poco más tarde, Tolo Garau, Francesc Bonin y Xisco Bonnin, también les agradezco su apoyo en este tiempo y la amistad que me han brindado. Mención aparte merece Manuel *o engenheiro* Barranco, con el cual he podido trabajar (y viajar) codo con codo, y que es una de las personas más nobles y trabajadoras que me he cruzado en la vida; espero que podamos seguir colaborando por muchos años. Finalmente, aun sin ser miembro oficial del SRV, le quiero agradecer a Sebastìà Roca su esfuerzo durante la realización del prototipo de OCS-CAN como parte de su PFC; su implicación y entusiasmo con el proyecto fueron clave para llevar a buen término esta parte del trabajo.

También me gustaría dar las gracias a los compañeros del *Departament de Ciències Matemàtiques i Informàtica*. En particular, a los miembros de las diferentes directivas: Arnau Mir, Pep Lluis Ferrer, Ricardo Alberich y Margaret Miró, y a los representantes en la Escuela Politécnica: director, secretarias/os, jefes de estudios... porque sé que su esfuerzo diario hace que muchas cosas puedan funcionar. Mi más sincero agradecimiento a Maria Fiol por su ayuda con los trámites administrativos de esta tesis; su eficiencia y su simpatía son sin duda el mejor *interface* que uno puede desear para tratar con el impredecible universo burocrático. También me siento afortunado por tener compañeros como Jairo, Mercè, Biel, Chus, Antonio E., Ana Belén, Felip, Esperança, Óscar, Joan Pons, Joan

Rigo, Tomeu Alorda, Cristina, Diana, Toni Buades, Rafel, Pere Palmer, Miquel Mascaró y un largo etcétera. Ellos hacen cada día de nuestro lugar de trabajo un sitio mejor, más agradable e intelectualmente estimulante. Gracias, por último, a Javier Martín, por hacer la convivencia diaria en nuestro despacho tan sencilla, agradable e interesante.

I would like to express my gratitude to the many colleagues that during this time helped me to understand which parts of my work could be improved and also helped me to overcome some of my own limitations; they definitely contributed to increasing the quality of this work. First, I remind some engaging conversations with Alan Burns, Guiem Bernat and Ian Broster, during my short stay at the University of York, that really sparked the fire of this research. Also, during my stay at the *Universidade de Aveiro*, I remember having many nice discussions with Luís Almeida, Joaquim Ferreira and Paulo Pedreiras about CAN and its dependability properties. Since then, they have been always providing very sincere and useful feedback for my research and this is a gift that must not be underestimated. By the way, I also feel *muito obrigado pola sua hospitalidade* while I was in Aveiro... It was a great time in my life.

I feel particularly indebted to Hans Hansson, from the Mälardalen University, since he was the person who taught me the rudiments of UPPAAL and he has been always very supportive of my research. I also thank Thomas Nolte, Thilo Sauter, Georg Gaderer, Nicolas Navet, Luís Pinho, Lucia Lo Bello and Paul Pettersson for the interesting feedback they have provided at different stages of this work. My gratitude goes also to the anonymous reviewers of the papers and book chapters we have written, since their points of view always leaded us to some kind of improvement. Last but not least, I would like to thank three colleagues from my department, Mercè Llabrés, Antonio E. Teruel and Gabriel Cardona, which usually help me to review the mathematical notation of my papers. Their patience and thoroughness has changed not only the aspect of my equations and propositions, but also my ideas about research; in a positive way, of course! Gabriel Cardona also introduced me to Ti*k*Z, a fantastic tool for drawing graphs with LaTeX that has been a blessing in my life.

Para acabar, sólo me queda agradecer a las personas que me han estado ayudando incluso desde antes de comenzar este trabajo. A mis amigos de siempre, Óscar, César, Fernando, Javi, Jose, Ángela y Teresa, por su bonita forma de entender la amistad. A mi familia en general por todo su apoyo, pero en particular a mis abuelas y abuelos, por su actitud ejemplar ante la vida y por todos los sacrificios que han hecho para que los demás estemos tan bien como estamos. A mis tíos Miguel y Pili, y a mi primo Miguel, por la forma en que me ayudaron mientras estuve con ellos en Vigo. A mis hermanos, Luis, Miguel, Carlos y Pablo, por su particular forma de ayudarme a mantener los pies en la tierra. Y por último, quiero dar las gracias de corazón a mis padres por todo lo que me han dado, que en su caso es realmente *todo*.

Contents

List of Figures

List of Tables

List of UPPAAL Listings

Chapter 1

Introduction

This chapter presents the goals of this dissertation, together with the organization of the document.

1.1 Research context

Roughly speaking, a distributed embedded system is a computer system constituted by a set of nodes that cooperate among them in order to fulfill a common goal, which usually implies the control of another system. Distributed embedded systems always rely on a network for communicating the nodes, since nodes need to exchange messages to make cooperation possible. The *Controller Area Network* (CAN) fieldbus [ISO93] is currently one of the most popular communication networks for distributed embedded systems.

Even though the CAN fieldbus was originally designed by Bosch for in-vehicle communication, it has become a de facto standard for a wide range of distributed embedded systems. CAN is nowadays used not only in vehicles, but also in factory automation, medical equipment and machine control, among many others [Ets01]. A very important reason for the success experienced by CAN is its instant bit monitoring, along with the mechanisms that are built upon this property [ISO93]. Such mechanisms include a number of useful error control capabilities as well as a mechanism for prioritized access to the medium, the so-called *CAN arbitration*, which guarantees bounded response time for any message broadcast, even in the presence of transient channel faults [TBW95, BBRN05, DBBL07]. During more than two decades, these properties proved to be very helpful for the development of myriads of non-critical distributed embedded systems [CiA].

Despite its uncontested success in so many application domains, the adoption of CAN for developing *critical* distributed embedded systems is still controversial. The main reason for this controversy is that CAN exhibits many limitations with respect to dependability. According to the literature [Tör95, FMD+00, APF02, SP07, PPA+09], the main dependability limitations of CAN are:

- Low bit rate
- Large and variable jitter
- Limited flexibility
- Limited error containment
- Limited data consistency
- Limited support for fault tolerance
- Lack of clock synchronization

Nevertheless, regardless of such dependability limitations, the enormous popularity of CAN makes it a very cost-effective technology. Due to this, substantial efforts have been carried out in order to overcome these dependability limitations and extend the applicability of CAN to more critical systems, at a reasonable cost. Many researchers have addressed these limitations and several solutions are already available (e.g. in [APF02, SP07, RGR98, BB01, PV03]). Integrating all these solutions into a single, comprehensive architecture is still a pending issue, which may not be straightforward [RNBP03a], but it seems that CAN may become a reasonable choice even for critical applications [PPA+09].

1.2 Problem statement

The motivation of our research is to help in this effort of improving the dependability properties of CAN. We do it by bringing into focus one of the dependability limitations that, in our opinion, merits more attention: *the lack of a clock synchronization service*. In the context of CAN, the problem of clock synchronization has been usually addressed only superficially, but we claim that this service actually plays a fundamental role for the achievement of many dependability attributes. A careful look into the solutions proposed to overcome the dependability limitations of CAN reveals that most of them do rely on the existence of a synchronized clock. In this sense, it can be said that clock synchronization is a "hidden" requirement for having dependable CAN-based systems.

Furthermore, many of the techniques proposed for improving the dependability of CAN impose some strict requirements on the clock synchronization service. These techniques assume that the system works with a clock synchronized with very high precision, in the order of a few μs, and assume that this clock is available to all the nodes of the system. This calls for the adoption of fault tolerance techniques, which should make the clock synchronization service reliable enough.

However, the proposals that are currently available for clock synchronization over CAN [RGR98, FMD+00, LA03, APSP07, SP07] do not provide a service that fulfills these requirements, especially in what concerns fault tolerance. Most of current solutions are designed for achieving high-precision clock synchronization, but the provision of fault tolerance has not been satisfactorily addressed: whenever fault tolerance techniques have been proposed, they have been only partially defined and they have not been properly evaluated.

For all these reasons, it is clear that there is room for a novel solution for clock synchronization over CAN; a solution that must specifically address the requirements imposed by the techniques proposed for improving the dependability of CAN, i.e. high precision, fault tolerance and low cost.

1.3 Our approach

In this dissertation we present the architecture and the evaluation of a novel solution for clock synchronization over CAN. This solution, which we have named the *Orthogonal Clock Subsystem for CAN* (OCS-CAN), is specifically designed to meet the requirements that are required to achieve dependability over CAN.

OCS-CAN pursues four properties: high precision, low overhead, fault tolerance and cost effectiveness. In order to achieve high precision, OCS-CAN is implemented in hardware. However, for reducing its overall cost, OCS-CAN has been conceived as an *orthogonal* subsystem: its architecture is based on an independent hardware circuit, the so-called *clock unit*, which is attached to each CAN node of the network and provides the intended clock synchronization service without requiring any further changes of the nodes. Moreover, for reducing the system overhead and indirectly reduce the cost of the solution, OCS-CAN implements a master/slave algorithm for clock synchronization, with master redundancy for providing tolerance to faults of the master.

Besides the discussion of the OCS-CAN architecture, a significant part of this dissertation deals with the formal assessment of the fault tolerance mechanisms of OCS-CAN. This assessment has been performed both *analytically* and by means of *model checking* with UPPAAL, which is a model-checking tool based on the theory of timed automata [BDL04]. The results obtained show that the fault tolerance mechanisms of OCS-CAN work as intended, and that a specific precision is guaranteed under the considered fault conditions. But, perhaps more importantly, this assessment also provides very valuable insights about the relationship that exists between the potential faults of the system and the achievable precision. This will prove that, in the presence of faults, OCS-CAN exhibits a *graceful degradation* of the precision, which is indeed a very interesting property for the design of dependable applications.

In our opinion, providing not only a suitable solution for clock synchronization in CAN, but also the

proper means to assess its performance (in this case the precision) against different fault conditions, is the way to follow in order to develop an adequate architecture for supporting dependable applications over CAN.

1.4 The thesis

This document presents the work we have conducted in order to demonstrate the truthfulness of the following thesis:

"It is possible to design a clock synchronization service that fulfills the requirements for implementing dependable applications over CAN. The suitability of the clock synchronization service will be measured in terms of three attributes: high precision, fault tolerance and cost effectiveness".

1.5 Main contributions

The work conducted in order to validate our thesis leaded to the following main contributions.

Study of the state of the art on clock synchronization for dependable CAN

This study consists of two parts. In the first part, we analyze the solutions proposed for improving the dependability of CAN and infer which requirements they impose on the clock synchronization service. In the second part, we study the available solutions for clock synchronization over CAN and check whether they satisfy such requirements. The result of this study is that each solution presents several drawbacks with regard to the considered requirements.

Design and prototyping of OCS-CAN

Given that none of the available solutions for clock synchronization over CAN completely satisfies the requirements for dependable CAN, we propose a novel solution that is specifically intended to fulfill the requirements imposed by the techniques suggested for improving the dependability of CAN. This solution is called OCS-CAN.

OCS-CAN is made up of a set of independent devices, called clock units, which are attached to the nodes of the system. Each clock unit is provided with a CAN controller for exchanging synchronization messages with the other clock units. Therefore, the set of clock units constitute an independent subsystem that is intended to provide the clock synchronization service regardless of the application executed by the nodes. We say that this subsystem in *orthogonal* because it has been conceived as

an add-on subsystem, which can be adopted without the replacement of any hardware component nor any change of the software executed by the nodes.

The scheme for clock synchronization adopted by OCS-CAN is master/slave because it reduces both the computation and communication requirements. However, a master/slave scheme introduces a single point of failure in the system. Therefore, the master should be replicated and a number of mechanisms will be defined for managing master redundancy and guarantee clock synchronization despite potential failures of the master. This fault-tolerant algorithm will take advantage of the inherent properties of the CAN network in order to reduce the computation and communication overheads.

In our architecture, the clock units are implemented in hardware because it reduces the response time of the subsystem, and therefore improves the precision. For reducing the cost of the solution, the clock units should be implementable with low-cost hardware. In this document, and as a proof of concept, we will discuss its implementation on a medium range FPGA.

Analytical assessment of the precision guaranteed by OCS-CAN in the presence of faults

The aim of this analysis is to assess the precision guaranteed by OCS-CAN in the presence of both channel and node faults, for the most likely scenarios. We suggest a mathematical framework based on piecewise linear functions for modeling the evolution of the clocks of the clock units. This framework also considers the clock adjustment actions performed by the clock units and their effect on the clocks.

The outcome of the analysis is a series of equations that allow us to quantify the precision degradation of OCS-CAN in the presence of faults. However, this kind of analysis becomes more complicated when different types of faults are combined, and it will eventually become unfeasible for the most complex scenarios. For this reason, we will have to complement our analytical assessment with the formal verification of OCS-CAN by means of model checking.

Model checking of the precision guaranteed by OCS-CAN in the presence of faults

Model checking is a formal verification technique that allows the user to determine whether a certain property is fulfilled by a model of the system or not [BK08]. In this work, we will use model checking for assessing the precision achieved by OCS-CAN in the presence of faults. Among the available tools for model checking, we will use UPPAAL, since it is a model checker based on the theory of *timed automata* [ACD93] that is extensively used for the formal verification of real-time systems [BDL04].

Model checking is a technique that first requires building a formal model of the system under verification. For the case of OCS-CAN, this model must include the properties of the CAN network as well as the properties of the clock units, including both the potential faults of the network and of the clock units. Thus, we will have to model features such as message broadcast, arbitration, channel

errors, etc. Given that these features are characteristic of many CAN systems, the techniques we will develop for modeling them will have broader applicability than just the formal verification of OCS-CAN. Due to this, these modeling techniques constitute a significant contribution of our research and will be thoroughly described.

The results obtained with model checking prove that the fault-tolerant master/slave algorithm of OCS-CAN guarantees a certain precision for different fault conditions. It also shows the precision degradation exhibited by OCS-CAN as the fault scenarios become more complex and severe, and therefore complements the analytical assessment mentioned above.

Exhaustive study of the patterns for modeling different types of computer clocks with timed automata and proposal of a number of new modeling patterns

As previously noted, the UPPAAL model checker is based on the theory of timed automata and, in principle, allows the modeler to specify the behavior of a system over time in a very natural way. However, the timed automata formalism also suffers from certain limitations, which we had to address during the model checking of OCS-CAN.

The main difficulties concern the modeling of *drifting* computer clocks and the modeling of the *clock adjustment* operations executed by the clock units. These two features cannot be directly modeled with the timed automata formalism, in principle. Therefore, we were forced to develop a number of novel modeling techniques that circumvented this limitation and made the formal verification of OCS-CAN possible. Although these techniques were specifically developed for OCS-CAN, they are applicable for the formal verification of many other distributed systems.

This generality gives more relevance to the work carried out. For this reason, in this dissertation we will include an exhaustive discussion on the patterns we have used for modeling the clock of the clock units with timed automata. This discussion will start with some modeling patterns that are widely known by system modelers, and will finish with those modeling patterns that have been specifically developed through our research.

1.6 Organization of the document

This document is divided into two parts. The first part, which is constituted by Chapter 2, Chapter 3 and Chapter 4, introduces the background concepts that are required for understanding the work presented in the rest of this dissertation. Chapter 2 discusses the basic notions concerning clocks and clock synchronization; Chapter 3 introduces the concept of model checking, the theory of timed automata and the UPPAAL model checker; and Chapter 4 discusses the main features of the CAN fieldbus from a dependability perspective.

The second part is constituted by the rest of the chapters, and discusses the specific work that has been carried out in order to fulfill the aims of this thesis.

Chapter 5 addresses the state of the art in clock synchronization over CAN. It discusses the requirements of the clock synchronization service, which are inferred from the solutions for solving the dependability limitations of CAN, and then it shows that none of the currently available solutions for clock synchronization does fulfill such requirements.

In Chapter 6 we present the architecture of OCS-CAN. This chapter thoroughly describes the techniques that have been adopted in order to fulfill the requirements of the clock synchronization service, and describes the internal structure of the clock unit. The development and testing of the first prototype of OCS-CAN is also briefly discussed.

Chapter 7 is devoted to the analytical assessment of OCS-CAN. In this chapter, we present a mathematical framework for modeling the behavior of the clocks of the Clock Units, and use this framework for assessing the precision guaranteed by OCS-CAN under different fault conditions.

Chapter 8 discusses the main difficulties that are encountered when modeling computer clocks by means of timed automata, and describes a series of modeling patterns that can be used for overcoming these limitations. Some of the presented modeling patterns have been specifically developed for the formal verification of OCS-CAN.

In Chapter 9 we address the model checking of OCS-CAN with UPPAAL. The entire formal model of OCS-CAN is described in detail, and the results of the formal verification are presented and discussed.

Finally, Chapter 10 summarizes the work presented in this document, remarks the main contributions of this dissertation, discusses the publications that resulted from the presented work, and provides some insight for further research.

Chapter 2

Background on clock synchronization

In this chapter we will review some basic concepts about clock synchronization. We will start by settling the basic terminology about clocks that will be used throughout this document. After that, we will discuss the main principles for implementing clock synchronization. Finally, we will present a simple taxonomy for classifying the different types of clock synchronization algorithms.

The aim of this section is not to perform an exhaustive discussion on the topic of clock synchronization, but just to introduce the notions that are required for understanding the work presented in the document. For further reading on clock synchronization, excellent surveys can be found in the literature, for instance in [Sch87, AP97, Kop97, VR01].

2.1 Establishing the basic terminology about computer clocks

This section describes the types of clocks that can be found in a computer system, and discusses their properties. It will be shown that actual clocks tend to diverge as time passes by and that, due to this, some kind of mechanism for clock adjustment is required.

2.1.1 Ideal clocks

The function of any clock is to measure time. Therefore, prior to any discussion on clocks, we should establish what *time* is, or more properly speaking, what model of time we will consider.

In this document we assume the Newtonian concept of time, as is usually done for computer systems. I.e., time is considered an external and continuous dimension that is perceived equally everywhere. This notion of time, which is often referred to as *real time*, can be represented by the set of positive real numbers. Any time instant belonging to this external dimension is usually denoted with the letter t.

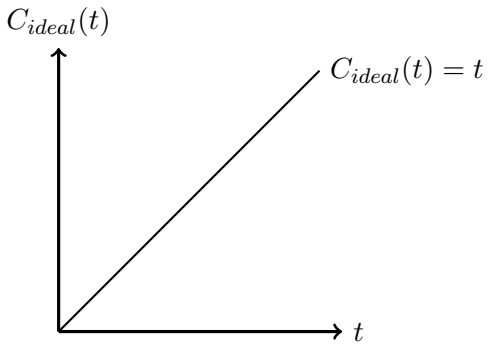


Figure 2.1: The value of an ideal clock always reflects real time

An *ideal clock* is the clock that always reflects the value of real time. Therefore, if we denote the value of an ideal clock at an instant t as $C_{ideal}(t)$, then $C_{ideal}(t) = t$ for every t. This concept is represented graphically in Figure 2.1.

It is important to remark that ideal clocks may not have physical existence. Therefore, they can be considered only theoretically, as an artifact to represent the evolution of real time.

2.1.2 Physical clocks

In order to physically measure real time, computers rely on devices that exhibit an approximately regular behavior over time: the local oscillators. A *local oscillator* is an electronic device that makes use of some kind of piezoelectric material, such as a quartz crystal or a ceramic resonator, to provide a periodic electrical signal of very precise frequency. Every pulse of the local oscillator is called a *microtick*.

Given that the nominal frequency of the local oscillator is known a priori, measuring time would be equivalent to counting microticks. However, since the frequency of the local oscillators is usually too high for what needs to be measured, the local oscillator is typically connected to a frequency divider. The output of the frequency divider is connected to a counter which should be externally accessible. The value provided by this counter is the value of the *physical clock*, which will be denoted as $C_i(t)$. Each pulse generated by the frequency divider is called a *tick* of the physical clock.

The typical scheme of a physical clock is depicted in Figure 2.2. It is constituted by the three elements mentioned above: the local oscillator (L.O), the frequency divider and the counter. Figure 2.2 also shows the output provided by each element.

Although the value of $C_i(t)$ increases in discrete steps (tick by tick), it can be satisfactorily approximated with a continuous function of rate $\dot{C}_i(t)$. However, local oscillators inevitably deviate from their nominal frequency (due to factors such as manufacturing imperfections, aging, tempera-

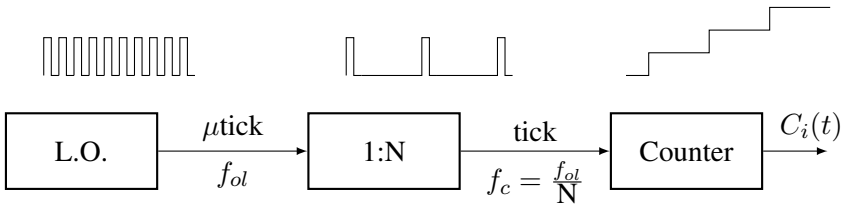


Figure 2.2: Scheme of the elements that constitute a physical clock

ture changes or variations of the power supply, for instance) and therefore the ticks of any physical clock can never happen at a perfectly constant rate.

The deviation between the rate of a physical clock and the rate of real time is called the *drift* of the physical clock, and it is obtained as follows:

$$\rho(t) = \frac{C_i(t) - C_i(t_0)}{t - t_0} - 1$$

A physical clock is said to be *non-faulty* as long as it exhibits a bounded drift. More rigorously, a physical clock i is said to be non-faulty if for every time instant t there is a value $\rho_i^{max} > 0$, such that

$$1 - \rho_i^{max} \leq \frac{C_i(t) - C_i(t_0)}{t - t_0} \leq 1 + \rho_i^{max}$$

Therefore, it is possible to express the rate of the physical clock in the form $\dot{C}_i(t) = 1 + \rho_i(t)$, with $\rho_i(t) \leq \rho_i^{max}$. The value of ρ_i^{max} is usually in the order of 10^{-4} to 10^{-6}.

Figure 2.3 shows the most common configuration for a node using a physical clock. In this case, the frequency divider and the counter (represented in the figure with the block labeled as Physical clock) are part of the processor, which means that they are implemented in software, at least partially. The application, which is also executed by the processor, uses the value provided by the physical clock for its computations.

In a distributed system, this configuration allows each node to work with its own (local) perception of time. However, given that each physical clock increases with a different drift, the nodes may not have a consistent perception of time, even if they use physical clocks with the same *nominal* rate. This is illustrated in Figure 2.4.

This figure shows the evolution over real time of two physical (and therefore drifting) clocks. For illustration purposes, the central line shows an ideal clock, which perfectly maps real time. The upper line shows a *fast* clock ($\rho_i > 0$) whereas the lower line shows a *slow* clock ($\rho_j < 0$). Notice that, since each physical clock exhibits an individual and different drift, their values tend to diverge, even if they had the same value initially.

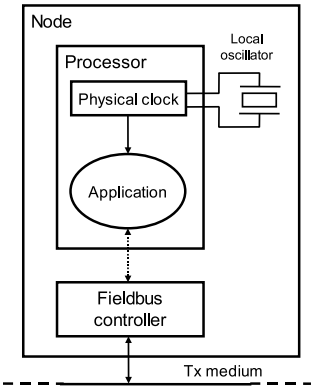


Figure 2.3: Structure of a node with a physical clock

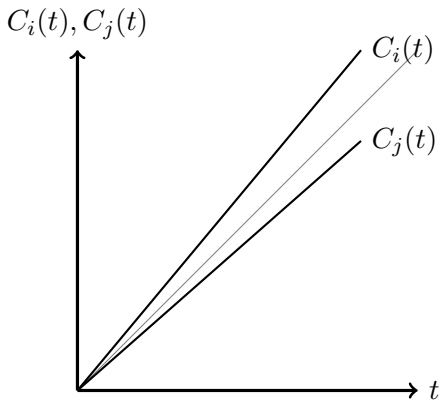


Figure 2.4: Behavior of two drifting physical clocks

At a given instant, the difference between the values of two clocks i,j is called their *offset* and is denoted as $\Phi_{ij}(t)$. In Figure 2.4 it can be noticed that the offset between $C_i(t)$ and $C_j(t)$ increases as time passes by. Therefore, it is obvious that this type of clocks do not guarantee that the nodes share a common perception of time.

2.1.3 Virtual clocks

Some distributed systems should be built upon the assumption that all the nodes have access to a time reference which is perceived equally everywhere within the system. The main advantage of having such a common time reference is that the nodes can rely on this reference in order to coordinate their operations. Nevertheless, this approach works only as long as every node is able to know the value of the common time reference very closely. This is a problem because each node only has access, in principle, to its local physical clock, and, as already discussed, physical clocks tend to diverge. For

this reason, an additional mechanism is required.

For solving this problem, each node also implements a periodical procedure, called the *clock synchronization algorithm*, which is intended to reduce the offset among the nodes and thus guarantee an approximately consistent perception of time. The clock that a node i obtains after applying the clock synchronization algorithm on its physical clock is called the *virtual clock* of node i, and will be denoted as $vc_i(t)$. Clock synchronization will be further discussed in Section 2.2 and Section 2.3.

The clock that provides the common time reference of the system is known as the *global clock*. An important characteristic of the global clock is that it does not map real time perfectly; it may diverge up to a certain extent. Moreover, it is important to remark that the global clock can have physical existence or not. The global clock does exist physically when it is chosen among the available virtual clocks in the system. This is the case, for instance, in the so-called master/slave approach for clock synchronization. In that approach, one of the virtual clocks (usually the one with the best-quality local oscillator) becomes the global clock, or *master*, and the other virtual clocks become *slaves* and try to follow the master's virtual clock as close as possible.

In contrast, the global clock does not exist physically when it is defined as a function of the values of a number of virtual clocks. For instance, for a certain time instant t, the value of the global clock $C_G(t)$ may be calculated by averaging the values of four virtual clocks as follows

$$C_G(t) = \frac{vc_1(t) + vc_2(t) + vc_3(t) + vc_4(t)}{4}.$$

The specific function that is used for calculating the value of the global clock is called the *convergence function*. Several convergence functions have been proposed in the literature (for instance, see [Sch87, AP97]) but discussing them is out of the scope of this chapter. Note that the rate of the global clock may be obtained in an analogous way, for instance by averaging the rates of a number of physical clocks.

2.2 Aims and phases of clock synchronization

Table 2.1 shows the types of clocks that have been discussed in Section 2.1, and summarizes their main properties. According to this classification of clocks, the function of the clock synchronization algorithm executed by each node is to get its own virtual clock as close as possible to the global clock.

The ultimate goal of this procedure is, however, to guarantee that the values of any two non-faulty virtual clocks of the system do not differ by more than a given amount Π, which is called the *precision*. This property, which is known as the *internal clock synchronization* property, can also be formulated as follows: for any pair of non-faulty nodes i, j and any time instant t, it is satisfied that $|vc_i(t) - vc_j(t)| \leq \Pi$.

Table 2.1: Types of clocks and their characteristics

Clock name	Other names	What it measures	Exists physically
Ideal clock	Newtonian clock, Perfect clock	The external and continuous dimension we call real time.	No, it is a theoretical construction.
Physical clock	Hardware clock	It counts ticks of a local oscillator, attempting to approximate the value of the ideal clock	Yes. It consists of a local oscillator, a frequency divider and a counter.
Virtual clock	Logical clock, Synchronized clock	It approximates the value of the clock defined as the time reference	Yes. It is a physical clock, plus a mechanism for clock correction.
Global clock	Absolute clock, Reference clock, System-wide clock	An absolute reference time, common to all nodes.	It may exist physically or not.

The quality of a clock synchronization algorithm is inversely proportional to the value of Π. The lower is the value of Π guaranteed, the better is the precision achieved.

For some distributed systems, maintaining internal clock synchronization is not enough. This is the case for instance of many large-scale distributed systems, in which the operation of the nodes need to be synchronized with some external source of time that provides a better approximation to real time, such as the Universal Time Coordinate (UTC). The fulfillment of this property is known as *external clock synchronization*. The aim of external clock synchronization is therefore to guarantee that the difference between any non-faulty virtual clock of the system and real time is less than a certain value α, which is called the *accuracy*, so that: $|vc_i(t) - t| \leq \alpha$.

A well-known result of clock synchronization states that guaranteeing an external clock synchronization of accuracy α enforces internal clock synchronization with precision $\Pi = 2\alpha$.

The clock synchronization algorithm is executed by each node individually, and it requires to periodically perform the following three phases [AP97]:

1. Detection of the periodic synchronization event. This event usually consists in the reception of a preconcerted and periodic message, although other alternatives exist.
2. Estimation of the value of the global clock at that specific time instant.
3. Calculation and correction of the error of the (local) virtual clock.

Figure 2.5 shows a simplified example of clock synchronization with three nodes, in which the exe-

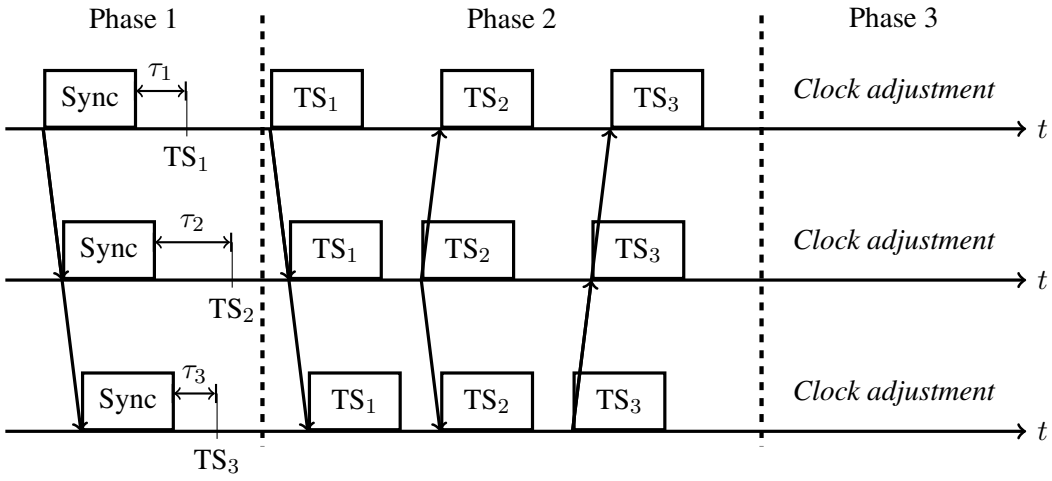


Figure 2.5: Phases of a clock synchronization algorithm

cution of these three phases is visible. The figure uses a different timeline for representing the actions performed by each node. In this example, the synchronization event is the reception of the message labeled `Sync`. This message is broadcast by one of the nodes, even though it is not represented in the figure. Due to the different network latencies, every node receives the `Sync` message at a slightly different instant. Then, as soon as a node receives the `Sync` message, it has to take a sample of its own virtual clock. This sample is called a *timestamp* of the `Sync` message and it is denoted as TS$_i$.

The processing of the just received `Sync` message needs some time, which should be different at each node. This delay is represented in Figure 2.5 with the delay τ_i. Thus, the combination of both latencies (the network latency and the processing latency) makes the nodes sample their virtual clocks at different real time instants. This difference causes a certain error that, as it will be discussed in Section 2.3, has a negative impact on the best achievable precision. Due to this, the separation among the timestamping instants should be reduced as much as possible.

In Phase 2, the nodes exchange the timestamps they have taken in Phase 1 and use them in order to obtain the value of the global clock. The number of nodes that send the timestamp depends on the specific type of clock synchronization being used. For instance, in master/slave clock synchronization, only one of the nodes sends its timestamp, and the other nodes take this value as the value of the global clock. Whereas in distributed clock synchronization, several nodes send their corresponding timestamps, and then each node applies a predefined convergence function (for example, the average function) on these values for obtaining the value of the global clock.

In Phase 3, also known as *clock adjustment*, each node calculates the difference between the value of its virtual clock and the (estimated) value of the global clock. Using this information, the node calculates a correction term that has to be applied for correcting the already accumulated offset with

respect to the global clock as well as for reducing the offset to be accumulated in the near future.

2.3 A simple taxonomy of clock synchronization algorithms

The wide variety of clock synchronization algorithms that are found in the literature is a consequence of the multiple ways to carry out each one of the phases discussed in Section 2.2. Since it is not possible to discuss every possible implementation of clock synchronization, in this section we will focus on the techniques that are particularly relevant for the design of OCS-CAN. More specifically, we will study some of the techniques that can be applied in Phase 1 and Phase 2 and will discuss how they affect the properties of precision, fault tolerance and cost effectiveness.

2.3.1 Software vs. hardware timestamp

As explained in Section 2.2, the first phase of any clock synchronization algorithm is detecting the synchronization event (the `Sync` message) and taking the associated timestamp. This phase can be implemented either in software or in hardware.

On the one hand, the choice between software or hardware has a strong impact on the precision, because the processing latency for detecting the `Sync` message (τ^i in Figure 2.5) is much higher for the case of a software-implemented mechanism. As indicated in ([Kop97], Table 3.2) whenever the timestamp is implemented in hardware, the achievable precision is in the order of a few μs, whereas for a software implementation the precision is only of about hundreds of μs.

On the other hand, the implementation of the timestamp mechanism in software is a better solution concerning cost-effectiveness. A hardware implementation requires installation of a specific circuit in every node that performs the detection of the `Sync` message. Due to this, software detection is, in principle, more flexible and also has lower installation costs.

2.3.2 Symmetric vs. asymmetric schemes

In Section 2.2, it was indicated that the number of messages to be exchanged in Phase 2 may vary depending on the implementation. According to the number of messages exchanged, it is said that Phase 2 follows either a *symmetric* scheme or an *asymmetric* scheme.

In a symmetric scheme, every node sends a message conveying its timestamp of the synchronization event. After that, each node applies the same convergence function in order to obtain the value of the global clock. This scheme takes takes into account, in principle, the contribution of each clock of the system. In contrast, in an asymmetric scheme there is one privileged node (the master), and it

is the only node allowed to send its timestamp. The rest of nodes (the slaves) use this timestamp to synchronize to the master.

Concerning precision and accuracy, both the symmetric and the asymmetric scheme can achieve the same values. Symmetric solutions are often considered more accurate since the use of many clocks compensates their drifts and results into a global clock that very closely approximates real time. Nevertheless, a master/slave scheme may also provide very good accuracy as long as the master has access to an accurate time source; for instance, if the master is provided with a GPS receiver.

In order to reach fault tolerance, the use of a symmetric scheme is more recommendable. Symmetric schemes are naturally redundant and hence favor implementation of more complete fault tolerance mechanisms. In contrast, in an asymmetric scheme the master constitutes a single point of failure of the system, since a failure of the master would left all of the nodes unsynchronized. Nevertheless, the master's reliability can also be improved by investing in its quality or by means of fault tolerance techniques such as master replication.

Concerning cost-effectiveness, the asymmetric scheme seems to be more suitable for distributed embedded systems with limited bandwidth available, since these schemes require transmission of much less messages. Moreover, symmetric schemes require nodes to manage a high number of messages in a short time interval, and this overhead can exceed the capacity of certain low-cost processors.

Chapter 3

Introduction to model checking and timed automata

This chapter introduces the basic knowledge that is required to understand the formal verification of OCS-CAN. Among the existing possibilities, the formal verification of OCS-CAN has been carried out by means of model checking, with a tool based on the theory of timed automata called UPPAAL [BDL04]. Therefore, this chapter will discuss the fundamentals of model checking, introduce the theory of timed automata, and present the basic features of UPPAAL.

3.1 Main techniques for system evaluation

There are three common ways to investigate the correctness of a system: *simulation*, *testing* and *formal verification*. Each one of these techniques exhibits some strengths and some weaknesses, which make them complementary [Kat98]. For this reason, they are usually combined in order to obtain the best out of each technique.

Simulation is based on the use of a model that describes the possible behavior of the system. This model is somehow executed with a software tool (known as the *simulator*) in order to observe the reactions of the system on certain stimuli. It is useful for a quick, first assessment of the quality of a design. But it is less suited to find subtle errors since it is infeasible (and often just impossible) to simulate all representative scenarios.

In *testing*, one takes the actual implementation of the system, stimulates it with certain inputs, observes its reactions (typically measuring the outputs) and checks whether they conform to the required outputs. The set of inputs used for stimulating the system are called the *tests*, and they should be carefully chosen as they should be representative enough. Nevertheless, testing can only show the

presence of errors, never their absence.

The aim of *formal verification* is to prove —in the strict mathematical sense of the word— that a system operates correctly; where correctly means that it does not deviate from its expected behavior. Thanks to this mathematical approach, formal verification (unlike simulation and testing) can provide guarantees about the absence of errors in a system. For this reason, this type of evaluation is particularly important for the design and assessment of fault-tolerant systems.

The procedure that must be followed for formally verifying a system consists of three tasks:

1. Constructing a formal (i.e. mathematical) model of the system under investigation, which must represent the possible behavior of the system.
2. Writing the correctness requirements in a formal requirement specification, which represent the desirable behavior of the system.
3. Finding a formal proof (a mathematical demonstration), which shows that the possible behavior *agrees with* the desired behavior.

There exist several techniques for formal verification, which can be grouped into two categories: *theorem proving* and *model checking* [Kat98]. Although both groups follow the procedure discussed above, the way to perform each task may vary from one technique to another. In the rest of this chapter, we will focus on model checking since it has been the technique adopted for the formal verification of OCS-CAN.

3.2 The concept of model checking

The idea behind model checking is to use algorithms, executed by computer tools called *model checkers*, to verify the correctness of systems. According to the scheme discussed in Section 3.1, this means that the last step (finding a formal proof) is left to the computer, whereas the user is still responsible for providing both a formal model of the system and a formal specification of the requirements (or properties) to be fulfilled. Figure 3.1 depicts the basic scheme of the model checking procedure.

Models are usually specified in the form of automata, whereas the properties are usually stated in some kind of temporal logic. Once they are provided, model checking is an automated technique that systematically checks whether the intended properties hold for the model [CGP01, BK08]. To do so, the model checker performs an *exhaustive state space search* of the model of the system: it generates all the possible states of the model and for each state it checks whether the property is satisfied or not. In its simplest form, this technique is known as *reachability analysis* [Kat98].

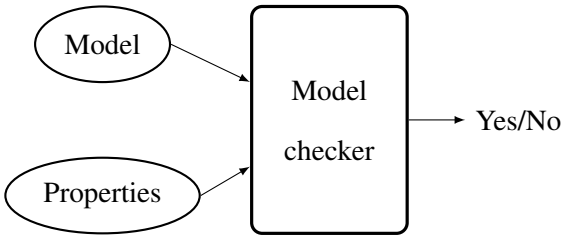


Figure 3.1: Model checking procedure

Moreover, whenever the system model does not fulfill the intended property, the model checker provides a counter example (in the form of a trace) to show under which circumstances the property is violated. This feature is one of the main strengths of model checking, since it makes it possible not only to find errors in the system design, but also to detect flaws in the formal model as well as in the requirements specification.

Model checking was proposed in the early 80's, in the pioneering works by Clarke and Emerson [CE82], and Queille and Sifakis [QS82]. Since then, this technique has been continuously investigated and improved in order to make model checking more efficient and thus suitable for larger hardware and software system. For instance, the development of *Symbolic Model Checking with Ordered Binary Decision Diagrams (OBDD)*, *Partial order reduction*, and *Bounded Model Checking with propositional satisfiability (SAT)* made it possible to apply model checking to complex systems (e.g. see [Kat98], Chapter 5). Current examples of the industrial application of model checking include verification of the designs for integrated circuits (particularly microprocessors) as well as communication protocols, software device drivers, real-time embedded systems and security algorithms, among many others [CGP01, BK08].

As a clear indicator of the success experienced by model checking, ACM (the Association for Computing Machinery) named Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis the winners of the 2007 A.M. Turing Award, for *their original and continuing research in the quality assurance process known as model checking* . This award is widely considered the most prestigious one in computing. The *Turing Lecture from the winners of the 2007 ACM A.M. Turing Award* can be found in [CES09].

Nevetheless, despite its several advantages, model checking has also certain limitations. A thorough discussion on these limitations is out of the scope of this work, but we would like to remark the two limitations that have more importantly affected the formal verification of OCS-CAN.

1. **The state-space explotion problem**. This is probably the most well-known limitation of model checking. It is related to the exhaustive state space search, and occurs whenever the full state space cannot be effectively computed by the model checker. This leads to *decidability* prob-

lems: for some systems it may not be possible to determine whether a given property is satisfied or not. This limitation is usually circumvented by abstracting away the features of the model that are irrelevant with respect to the properties to be verified. However, applying the right abstractions is not straightforward and requires some expertise.

2. **Model vs. implementation**. By means of model checking one does not check the actual implementation of the system, but only a model of it. Therefore, the suitability of this technique strongly depends on the capacity of the user to specify a model that captures the real behavior of the system under verification.

Because of these two limitations, the use of model checking typically implies a compromise between complexity and realism. Too much realism (i.e. specifying a too detailed model) may cause excessive complexity and may lead to decidability problems, whereas too less realism may lead to wrong or, at least, irrelevant verification results.

For the case of a real-time system, a *realistic* specification must include not only the functional aspects of the system, but also its temporal behavior. Fortunately, certain formalisms have been developed for model checking, which allow specification of temporal aspects without introducing excessive complexity. For instance, the notion of *timed automaton* [ACD93], to be discussed in Section 3.3, extends the idea of finite state automaton with a quantitative representation of time that has been very useful for model checking real-time systems.

3.3 The theory of timed automata

A timed automaton is basically a discrete automaton extended with a finite set of real-valued variables, named *clocks*, for modeling the passage of time. In this section, we introduce the formal definition of timed automaton and discuss its dynamical behavior with a simple example.

3.3.1 Clocks and clock constraints

Before introducing the concept of timed automaton, we have to define the concepts of clock and clock constraint, as they constitute a fundamental part of the timed automaton formalism.

Definition 1 (Clock)**.** *A* clock *is a variable ranging over* $\mathbb{R}^+$.

Clocks are usually denoted with the letters x, y, z. A *clock constraint* is a condition defined over a clock x or over a clock difference $x - y$.

Definition 2 (Clock constraint)**.** *For a set C of clocks, with $x, y \in C$, the set of* clock constraints *over C, $\Psi(C)$, is defined by*

$$\alpha := x \prec c \mid x - y \prec c \mid \neg\alpha \mid (\alpha \wedge \alpha)$$

where $c \in \mathbb{N}$ and $\prec \in \{<, \leq\}$.

For the sake of brevity, clock constraints of the form $\neg(x < c)$ and $\neg(x \leq c)$ will be written, respectively, as $x \geq c$ and $x > c$.

3.3.2 Formal definition of timed automaton

The notions of clock and clock constraint are incorporated to the timed automaton formalism as follows [Kat98].

Definition 3 (Timed automaton)**.** *A* timed automaton *A is a tuple $(L, l_0, E, Label, C, clocks, guard, inv)$ with*

- *L, a non-empty, finite set of locations with initial location $l_0 \in L$*
- *$E \subseteq L \times L$, a set of edges*
- *Label : $L \rightarrow 2^{AP}$, a function that assigns to each location $l \in L$ a set Label(l) of atomic propositions*
- *C, a finite set of clocks*
- *clocks: $E \rightarrow 2^{C}$, a function that assigns to each edge $e \in E$ a set of clocks, clocks(e), to be reset*
- *guard : $E \rightarrow \Psi(C)$, a function that labels each edge $e \in E$ with a clock constraint guard(e) over C, and*
- *inv : $L \rightarrow \Psi(C)$, a function that assigns to each location an invariant inv(l).*

Note that this definition is very similar to the usual definition of discrete automaton. However, there are two important differences between both types of automata.

1. The definition of state in a timed automaton changes from the usual definition of state in a discrete automaton. In a timed automaton, the *state* consists of the current *location* of the automaton plus the current values of *all* clock variables. In principle, this makes the number of possible states of a timed automaton uncountable. However, during the model checking process the states can be grouped so as to form a finite set; the automaton obtained as a result of this grouping process is called the equivalent *transition system* [ACD93, Kat98]. The reachability analysis is feasible for this transition system, what makes model checking possible.

2. A discrete automaton may change its state only in one way, by means of a discrete transition. In contrast, a timed automaton may evolve in two different ways. The first one, which coincides with the discrete automaton's only way, is called a *discrete transition* and occurs whenever the timed automaton traverses an enabled edge and changes to a different location. These transitions take no time. The second way, specific for timed automata, is called a *delay transition*. A delay transition causes all the clocks of the timed automaton to increase by an infinitesimal quantity, while staying in the same location.

In summary, the initial state of a timed automaton is the pair (l_0, v_0), where l_0 is the initial location of the timed automaton, and v_0 is the time assignment assigning 0 to all clock variables. Once initialized, clocks start incrementing their value implicitly, and they all proceed at the same rate. Conditions on the values of clocks (i.e. clock constraints) are used as enabling conditions (or *guards*) of discrete transitions: only if the clock constraint is fulfilled, the transition is enabled, and can be taken; otherwise, the transition is blocked.

Clocks can be initialized only when the system makes a discrete transition. The value of a clock thus denotes the amount of time that has been elapsed since it has been initialized. Invariants on clocks are used to limit the amount of time that may be spent in a location and to force the model to progress over time.

3.3.3 Dynamics of a timed automaton

In order to illustrate the dynamics of a timed automaton, let us consider the timed automaton A_1 shown in Figure 3.2. For depicting timed automata we adopt the following convention: circles denote locations and edges are represented by arrows. The initial location is highlighted with a thicker line.

The timed automaton of Figure 3.2 has two clocks, x and y, two locations l_0 and l_1, and an edge from location l_0 to l_1 labeled with the guard $y \geq 1$, and the reset set $\{x\}$. Location l_0 is labeled with the location invariant $x \leq 2$. In the following, invariants will be written in bold face in order to differentiate them from guards.

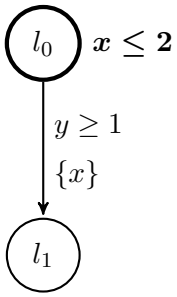


Figure 3.2: Timed automaton A_1

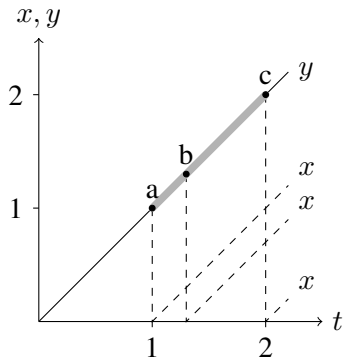


Figure 3.3: Possible behavior of timed automaton A_1, with three potential traces

Assuming that all of the clock variables are initially set to zero and that the initial control location is l_0, the automaton starts in the state $(l_0, x = y = 0)$. As the clocks increase (by means of delay transitions) synchronously with time, it may evolve to all states of the form $(l_0, x = y = t)$, where t is a non-negative real number less than or equal to 2. At any state with $t \in [1, 2]$ it may change to state $(l_1, x = 0, y = t))$ by following the edge from l_0 to l_1, that resets x.

Figure 3.3 shows over a single graph three possible traces of clocks x and y, according to A_1. The grey area indicates the interval $t \in [1, 2]$ in which a discrete transition from l_0 to l_1 is possible. The points labeled as a, b and c correspond to the instants $t = 1$, $t = 1.2$ and $t = 2$, respectively. For each point, a dashed line indicates the behavior of clock x if the transition would be taken. Note that the timed automaton may not idle forever in location l_0 since the location invariant $x \leq 2$ forces it to leave location l_0 within 2 time units.

3.4 The challenge of modeling computer clocks with timed automata

The main advantage of the notion of clock included in the timed automata theory is that it models the passage of time in a quantitative manner. This allows both the specification and the verification of quantitative temporal properties, such as "the automaton stays in location l not longer than t time units" or "property ϕ holds for t time units", to give just some simple examples.

The notion of time featured by these clocks has proved to be especially useful for model checking real-time systems, since the temporal requirements of this kind of systems are mainly specified quantitatively, e.g. in the form of periods, deadlines, etc. Nevertheless, it is also known that the dynamics of the clocks of a timed automaton are sometimes too idealistic and that for certain systems, typically distributed systems, these clocks may fail to reflect their real temporal behavior [ATM05].

In this section we will discuss what causes this limitation of the timed automata theory and how it relates to the problem of model checking OCS-CAN. Later on, in Chapter 8, we will describe the modeling techniques that will be applied for circumventing this limitation and formally verifying our system.

3.4.1 Some remarks about our nomenclature

At this point it is interesting to review the nomenclature we established in Chapter 2 for designating each type of clock that can be considered in a distributed system. In Chapter 2, we indicated that the nodes of a distributed system may work with three types of clocks: ideal clocks, physical clocks and virtual clocks.

According to their implementability, these three types of clock can be grouped into two categories. Ideal clocks belong to the category we define as *theoretical clocks*, given that they are an abstract notion that cannot be implemented. In contrast, both physical clocks and virtual clocks belong to the category we define as *computer clocks*, given that they can be actually implemented as devices, i.e. by means of hardware and software. Computer clocks are therefore subject to imperfections, aging and other phenomena that make their behavior far from being ideal.

In Section 3.3, we indicated that a timed automaton (TA, for short) is basically a discrete automaton extended with a number of real-valued variables, named *clocks*. In order to differentiate these clock variables from the devices known as computer clocks, we will hereafter refer to the former as *TA clocks*.

In the following discussion, we will explain why TA clocks cannot be used directly for modeling computer clocks. Basically, it will be shown that TA clocks are built upon certain assumptions that do not hold for computer clocks.

3.4.2 Temporal evolution of a set of TA clocks

As indicated in Section 3.3.3, the semantics of a TA assumes the TA clocks to be continuous and to evolve at the constant rate of *real time*. This corresponds to the notion of ideal clock, as introduced in Chapter 2. However, the devices that computer systems use for measuring time, i.e. the *computer clocks*, exhibit a non-ideal behavior. On the one hand, physical clocks always exhibit certain drift with respect to real time, and also with respect to each other. On the other hand, even when some kind of mechanism for clock synchronization is applied on the physical clocks, the virtual clocks obtained still evolve at slightly different rates, since perfect clock synchronization between virtual clocks is unfeasible in practice.

In Section 2.1, we specifically characterized the behavior of the computer clocks with two variables: the value and the rate. We also defined a measure for characterizing the synchronization between a pair of computer clocks: the *offset*, which is the difference between the clocks' values. These concepts can also be used for explaining the problem of realistically modeling computer clocks by means of TA clocks.

According to the theory of TA, when the operation of a TA starts, at $t = 0$, the value of each TA clock equals 0. From that moment on, each TA clock x increases at a constant rate $\dot{x} = 1$, which obviously means that $x = t$. However, if a TA clock x is reset at a certain instant, for instance at $t = t_x$, then the value of the TA clock will be $x = t - t_x$ from that moment on, until it is reset again.

Figure 3.4 illustrates the implications that this behavior has on the offset between the TA clocks. This figure shows the evolution of two TA clocks, x and y, that are reset at t_x and t_y, respectively. Given that these TA clocks belong to the same automaton, they increase at the same rate. This is noticeable in the fact that they evolve following parallel lines.

Figure 3.4 also shows that the offset between the TA clocks changes as a consequence of every reset of a TA clock, but remains unchanged otherwise. In fact, the offset between x and y at any instant t is a constant that equals the distance between the instant in which y was reset for the last time and the instant in which x was reset for the last time, since $x - y = t - t_x - (t - t_y) = t_y - t_x$. For the specific case depicted in Figure 3.4, the offset between the TA clock x and the TA clock y is:

$$\begin{cases} x - y = 0, & \text{for } 0 \leq t < t_x; \\ x - y = -t_x, & \text{for } t_x \leq t < t_y; \\ x - y = t_y - t_x, & \text{for } t \geq t_y. \end{cases}$$

This behavior does not correspond to the typical behavior of a pair of computer clocks, as it is discussed in the next subsection.

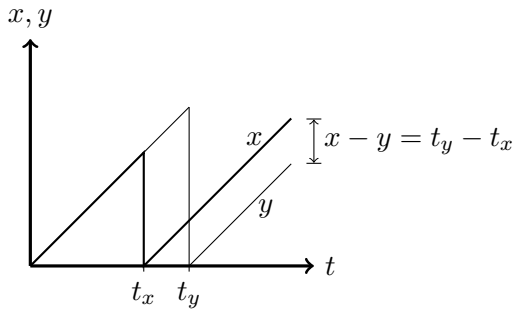
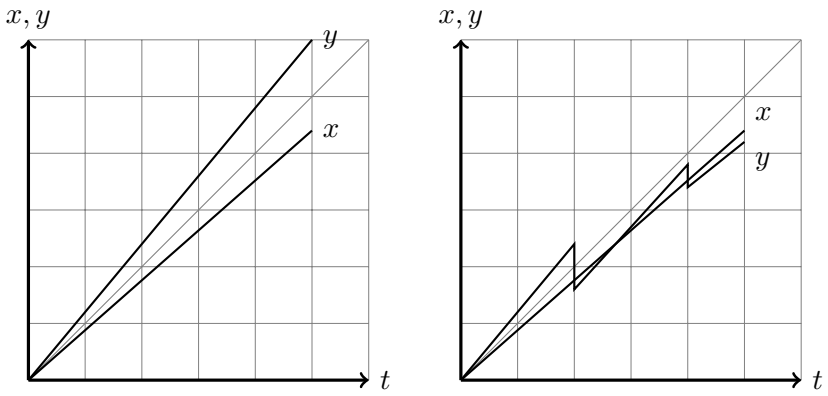


Figure 3.4: Representation of the temporal evolution of two TA clocks belonging to the same TA



(a) (b)

Figure 3.5: Possible temporal evolution of two computer clocks: (a) depicts the evolution of two physical (drifting) clocks; (b) depicts the temporal evolution of two virtual (synchronized) clocks

3.4.3 Temporal evolution of a set of computer clocks

Figure 3.5 depicts the typical temporal evolution of a pair of computer clocks, for the two cases possible. On the one hand, Figure 3.5(a) represents the evolution of two physical clocks that drift apart. On the other hand, Figure 3.5(b) represents two virtual clocks that are synchronized such that virtual clock y is periodically corrected in order to follow virtual clock x.

If we compare the behavior of these computer clocks with the temporal behavior of a pair of TA clocks, which was shown in Figure 3.4, we will see that there exist two significant differences:

1. Both physical clocks and virtual clocks evolve at a rate different from real time. Since each computer clock increases at a different rate, the offset between them is not constant but keeps increasing with time, unless (for the case of the virtual clocks) some kind of clock correction

is applied. Notice that, in contrast to the behavior exhibited by TA clocks, the fact of being initially synchronized with offset 0 does not imply that the computer clocks remain synchronized.

2. For the case of a pair of virtual clocks, both the value and the rate of a given clock may change abruptly in order to follow the value and the rate of another virtual clock.

The specification of these two characteristics will constitute the main challenge for the realistic modeling of computer clocks with timed automata. Fortunately, even though computer clocks cannot be directly modeled by means of TA clocks, it is possible to apply certain modeling patterns, which will overcome the limitations of the TA clocks and allow the specification of systems conforming to the behavior of a system that uses computer clocks.

These modeling patterns will rely on two specific techniques. The first technique is the one known as *perturbed timed automaton*, which was first described in [ATM05]. The second technique has been specifically developed for the formal verification of OCS-CAN, and is therefore a significant contribution of this work. We have coined the term *clock pointers* for designating this technique. The concept of perturbed timed automaton will be discussed in Section 3.5, but the concept of clock pointer will be discussed in Chapter 8, when describing the modeling patterns for systems with computer clocks.

3.5 The concept of perturbed timed automaton

The aim of this section is to introduce a new class of automaton, the so-called *perturbed timed automaton*, which will be fundamental in order to model systems with physical and virtual clocks. The notion of perturbed timed automata was defined in [ATM05] as a means to specify a particular kind of hybrid systems: those that can be modeled as rectangular hybrid automata [HKPV98]. However, addressing the topic of formal verification of hybrid systems is out of the scope of this document, and hence we will only discuss those details that are required for understanding the modeling patterns of Chapter 8.

A perturbed timed automaton is also a discrete automaton with some real-valued clocks, but the semantics of a perturbed timed automaton allows one of the clocks to vary within a certain constant range $[x_l, x_u]$, such that $x_l \leq \dot{x} \leq x_u$. Therefore, a timed automaton as defined in Section 3.3, can be seen as a trivial case of perturbed timed automaton in which $1 \leq \dot{x} \leq 1$ for every clock x.

In [ATM05], it is shown that any perturbed timed automaton can be translated into an equivalent timed automaton, as long as the bounds over the clock rate (x_u and x_l, mentioned above) are constant and known a priori. This result is important because it implies that the class of perturbed timed automata belongs to the class of automata that can be verified by means of model checking [HKPV98].

The rationale behind the translation from perturbed timed automata to timed automata is that having a clock of bounded rate is equivalent to having an ideal clock and defining the occurrence instants as time intervals whose lengths depend on the rate boundaries. Therefore, to obtain a timed automaton, the modeler must place the uncertainty of the clock rate into the guards and invariants.

We will explain this transformation with a simple example. Let us consider a timer that expires after T time units, and let us consider that this timer uses a physical clock that ticks with a rate in the range $[1-\rho, 1+\rho]$. Note that this is consistent with the definition of physical clock given in Section 2.1.2.

Figure 3.6 shows the effect that the physical clock's drift may have in the expiration instant of the timer. The upper line depicts the evolution of a clock evolving at the highest rate allowed, whereas the lower line depicts the evolution of a clock evolving at the lowest rate allowed. These two lines then define an *envelope* for the potential behavior of the physical clock, which makes it possible to determine the earliest and the latest instants in which the timer may expire. These instants are, respectively, $t_1 = \frac{\mathrm{T}}{1+\rho}$ and $t_2 = \frac{\mathrm{T}}{1-\rho}$. The grey area in the abscise (between t_1 and t_2) corresponds therefore to the dense time interval in which the expiration is possible.

The end points of this interval are the key for the transformation into timed automata, because the same behavior can be achieved if we build the model with a timed automaton that has a guard condition for the earliest occurrence instant and a time invariant for the latest occurrence instant. The resulting automaton is depicted in Figure 3.7(a). Assuming that the transition from location l_0 to location l_1 is the event that the timer triggers, Figure 3.7(a) shows that the temporal behavior of the timed automata would be the same as the one of the perturbed timed automata, provided that $\mathrm{T}_{max} = t_2 = \frac{\mathrm{T}}{1-\rho}$ and $\mathrm{T}_{min} = t_1 = \frac{\mathrm{T}}{1+\rho}$.

However, and due to the fact that $\rho \ll 1$, in our modeling we will apply these appromixations: $\mathrm{T}_{max} = \frac{\mathrm{T}}{1-\rho} \simeq \mathrm{T}(1+\rho)$, and $\mathrm{T}_{min} = \frac{\mathrm{T}}{1+\rho} \simeq \mathrm{T}(1-\rho)$. The main reason to use these approximations is to facilitate the work with UPPAAL, because this tool only allows definition of guards over integer values, and it is easier to be guaranteed with a multiplication than it is with a quotient.

It is important to remark that, when applying this modeling technique, the resulting timed automaton will include all the possible behaviors for every clock rate in the range $[1-\rho, 1+\rho]$. This constitutes a very positive aspect of the modeling, because it guarantees that the results provided by model checking are not only valid for one specific clock rate, but for the whole range of possible rates. This is a significant advantage in front of testing, for instance, in which generating all the possible clock rates is not possible.

The notion of perturbed timed automata and the transformation into timed automata will be of paramount importance in order to model the computer clocks of OCS-CAN. The way in which this modeling technique should be applied will be further discussed in Chapter 8.

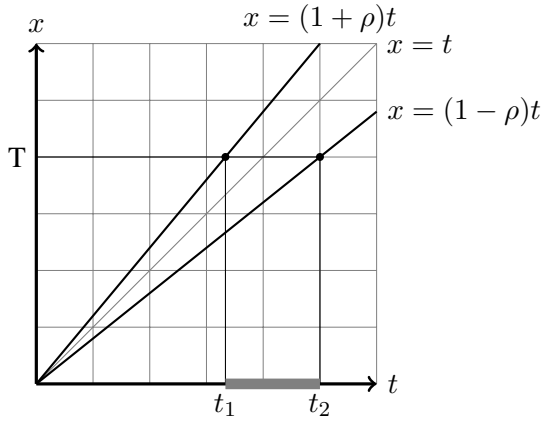
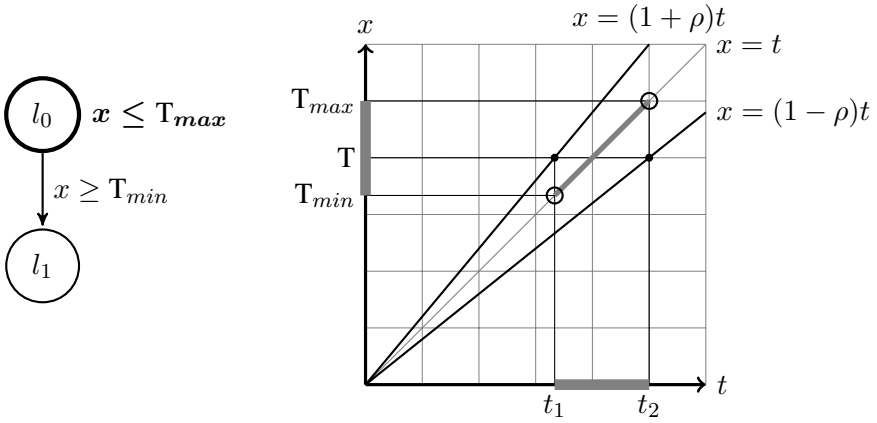


Figure 3.6: Graphical representation of the potential uncertainty caused by the physical clock's drift



(a) (b)

Figure 3.7: Example of a translation to timed automata: (a) is the resulting timed automata, (b) is the graphical description of the translation

3.6 The UPPAAL model checker

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems [BDL04, LPY97, BDH+06]. It was jointly developed, and is jointly maintained, by the Department of Information Technology at Uppsala University, in Sweden, and the Department of Computer Science at Aalborg University, in Denmark. It was first released in 1995, and since then is has become probably the most widespread model checking tool based on the theory of timed automata [BK08].

More specifically, UPPAAL allows the modeling of systems as *networks of timed automata*, extended with certain data types, such as bounded integers and arrays. In addition, UPPAAL not only performs the verification of the properties of a system model, but also integrates a tool for creating the models to verify by means of a user-friendly Graphical User Interface (GUI). UPPAAL also includes a simulator that allows the user to partially test her models before starting the automated verification of properties.

This section will discuss the modeling features of UPPAAL, together with the simulator and the verifier that the tool incorporates.

3.6.1 Networks of timed automata

A network of timed automata $\bar{A}$ is the parallel composition $A_1 \mid \ldots \mid A_n$ of a finite collection $A_1, \ldots, A_n$ of timed automata for a given synchronization function $|$. The UPPAAL tool uses networks of timed automata composed with a CCS-like parallel composition operator [Mil89]. It allows for individual components to perform internal actions (i.e. interleaving), and for pairs of components to synchronize on actions.

To describe the notion of a network of timed automata more formally, we define the state of a network automata $\bar{A}$ as a pair $(\bar{l}, v)$, where $\bar{l}$ is a location vector in $\bar{A}$ and v is a clock valuation over all clocks in $\bar{A}$. We further define the set of actions over alphabet Σ to be $\{a! \mid a \in \Sigma\} \cup \{a? \mid a \in \Sigma\} \cup \tau$, where τ is a distinct silent action.

A network of timed automata (like a single timed automaton) can only perform two types of transitions: discrete transitions and delay transitions. Nevertheless, it is possible to distinguish two kinds of discrete transitions, namely τ-transition and synchronization transition, depending on the number of automata involved.

On the one hand, a τ-transition only involves one timed automaton of the network. It occurs whenever a timed automaton A_i traverses an enabled edge that causes a change of location and potentially one or more clock resets. On the other hand, a synchronization transition occurs if there are two enabled edges with complementary synchronization actions, e.g. $a!$ and $a?$, that are traversed in two

different automata of the network. This transition causes a simultaneous change of location in each automaton involved, plus the potential reset of one or more clocks.

The delay transition is equivalent to the one discussed in Section 3.3. But, for a network of timed automata, it is important to remark that this kind of transition is possible as long as the conjunction of the location invariants of all automata is satisfied.

3.6.2 Modeling with UPPAAL

The term used by UPPAAL in order to designate a timed automaton is *process*. For this reason, it is said that a UPPAAL model is built as a set of *concurrent processes*. However, both terms, process and timed automaton, are equivalent and will be used interchangeably.

A UPPAAL process is represented as a graph which has locations and transition (edges) between locations. The firing of each transition may depend on a guard and/or a synchronization. *Guards* in UPPAAL are expressions defined over the variables and clocks of the model, so they are not only clock constraints. A *synchronization* corresponds to the parallel composition operator for networks of time automata discussed in Section 3.6.1.

Guards and synchronizations are not the only attributes of transitions. Transitions may also have associated actions that are executed when the transition is taken. Said actions are in the form of the *update* of some variables or clocks. For the case of clocks, they can only be updated to the value 0, which corresponds to the reset action presented in Section 3.3.

In order to define a synchronization for a transition, a *channel* must be defined in the model. Complementary synchronization actions are denoted $a!$ and $a?$, as it is common in networks of timed automata.

The modeling language of the UPPAAL tool features a number of extensions to the concept of timed automaton discussed in Section 3.3. The most important extensions are:

- Local and global bounded integer variables,
- Committed locations, and
- Broadcast synchronization.
- Urgent locations and channels

Bounded integer variables are simply data variables declared over bounded domains that can be assigned integer values in their domain. UPPAAL supports integer arithmetic expressions over data variables, constants, and the arithmetic operators $+$, $-$, $*$, and $/$. Whereas local variables are visible

only for a specific process, global variables are visible for every process of the network. Due to this, global variables may be used in combination with synchronization channels in order to model the passing of parameters among processes.

Committed locations are used to model atomic behaviors in UPPAAL by declaring a subset of the locations of an automaton as committed (denoted ⓒ or with prefix c:). If in a state $(\bar{l}, v)$ some location $l_i \in \bar{l}$ is committed, then the next transition of the network must be a discrete transition involving automaton A_i, otherwise the network is deadlocked. It is not possible to delay in a committed location.

Broadcast synchronization has been introduced in UPPAAL in order to extend the pair-wise synchronization described in Section 3.3. It requires definition of a so-called *broadcast channel*. The broadcasting synchronization involves one sender performing an action transition labeled $c!$, where $c \in \Sigma$, and an arbitrary number of receivers performing complementary action transition $c?$. Any process (except the sender) that has an enabled transition labeled $c?$ *must* take part in the broadcast synchronization. The number of receivers can be zero or more, hence, the sender can execute the $c!$ action transition even if there are no receiving automata. This is a significant difference with respect to the pair-wise synchronization, which is blocking.

UPPAAL defines the concept of *urgent location* and *urgent channel* as a means to better specify the dynamics of the model. Urgent locations are locations in which time is not allowed to pass. An urgent channel indicates that it is not possible to delay in the current state as long as a synchronization can be triggered through the urgent channel.

3.6.3 The simulator

Once the model has been built using the features described above, UPPAAL allows the user to simulate the behavior of the model and, thereby, detect both modeling mistakes and design errors in the modeled system.

The simulator graphically displays in a window all the processes included in the model, and continuously indicates which location is active in each process by placing a red dot in those locations. The simulator also shows in a window the value of all variables and in another window in which a temporal representation of the evolution of the model —including synchronization among processes— is built in real-time during the simulation. Finally the simulator records a trace of the performed simulation which can be saved, replayed and modified afterwards.

The simulator starts with the red dot of each process in the location configured during the modeling process as initial. By clicking on any one of the items in the list of enabled transitions, the corresponding transition is fired, the red dot passes to the next active location, variables are updated, and the transition is represented in the temporal diagram and added to the current trace. By choosing

different items in the list of enabled transitions it is possible to test different executing paths of the model. Note that transitions from different processes can interleave. Choosing different transitions among the enabled ones, different interleavings can be explored.

3.6.4 The verifier

Once the model has been analyzed and debugged using the simulator, it is ready for the last phase which is the verification. For this phase UPPAAL uses a *model-checker engine* which acts as a server of a client-server architecture and can be executed in a different computer than the GUI for performance improvement. There is also a stand-alone version of the engine that is interfaced through a command line.

The model-checker engine tests whether a series of properties defined by the user hold in the dynamic behaviors of the model. As already indicated, the main idea behind model checking is to build the full state space of the model and to exhaustively check if the intended properties are satisfied or not.

When the model-checker engine is interfaced through the GUI, the UPPAAL window is divided into several horizontal sections. In the upper section, the list of expressions (or *queries*) appears. Each query corresponds to one property to be checked. if the corresponding property holds, a bullet next to the query turns green, and if it does not hold, the bullet turns red. The model-checker engine can be requested to return a diagnostics trace when a property does not hold. This trace is essentially an execution path in which the property is violated, and that can be executed step by step using the simulator.

The next section of the window allows the editing of the query selected in the list above. Likewise, the next one allows the addition of comments to each query. Finally, the lower section of the window provides status information about the operation of the engine.

The types of properties that can be directly checked using the UPPAAL queries are quite simple. UPPAAL designers have taken the approach of not allowing complex queries, in order to improve the efficiency of the tool. For this reason, the verification of complex properties may need the checking of many different queries and even the addition into the model of specifically designed *observer* automata.

The specific types of properties that can be expressed in the UPPAAL query language can be classified as:

- *Reachability properties*: Any property of this class tests whether a specific condition —which is a boolean condition over locations, variables and clocks— holds in some state of the potential

behaviors of the model. This class of properties are always expressed in the form:

  - E $\diamond$ p, read as "exists eventually p", which tests if there exists an execution path in which p eventually —i.e. in some state of the path— holds. The notation used for this kind of expression in UPPAAL is E<> p.

- *Safety properties*: Properties of this class test whether a specific condition holds in all the states of an execution path. In fact there are two possibilities for this kind of properties, which are expressed in the following forms:

  - E $\Box$ p, read as "exists globally p", which tests whether there exists an execution path in which p holds for all the states in the path. The notation used for this kind of expression in UPPAAL is E[ ] p.
  - A $\Box$ p, read as "always globally p, which tests if for every execution path, p holds for all the states in the path. The notation used for this kind of expression in UPPAAL is A[ ] p.

- *Liveness properties*: Properties of this class test whether a specific condition is guaranteed to hold eventually. There are again two possibilities for this kind of properties, which are expressed in the following forms:

  - A $\diamond$ p, read as "always eventually p", which tests if for every execution path, p holds for at least one of the states in the path. The notation used for this kind of expression in UPPAAL is A<> p.
  - $q \rightsquigarrow p$, read as "q always leads to p", which tests if every execution path that starts from a state satisfying q reaches later a state in which p holds. The notation used for this kind of expression in UPPAAL is $q - - > p$. This type of liveness property can be written as A $\Box$ ($q \rightarrow$ A <> p), whose notation in UPPAAL is A[ ] (q imply A<> p).

- *Deadlock properties*: A state is in a *deadlock* if it is impossible that the model evolves to a successor state neither by the effect of an invariant nor by a transition between locations. Properties of this class test if a deadlock is possible or not in the model. UPPAAL allows the use of the reserved words *deadlock* and *notdeadlock* to check this properties. For instance, E<> deadlock means "it exists deadlock" whereas A[ ] notdeadlock means "there is no deadlock".

Chapter 4

The Controller Area Network from a dependability perspective

As already indicated in the Introduction, the CAN fieldbus is a de-facto standard for the design of non-critical distributed embedded systems. However, its suitability for critical applications is still controversial because of certain limitations with respect to dependability. This chapter introduces the main characteristics of CAN and discusses these dependability limitations.

Although in this dissertation we will only address one specific dependability limitation of CAN, the lack of a clock synchronization service, it is important to understand the general context of the work, which is the improvement of the dependability characteristics of CAN as a whole [PPA+09]. Moreover, as it will be shown in Section 5.1 and Section 5.2, the solutions proposed for solving some of the dependability limitations presented in this chapter will influence the requirements to be fulfilled by the clock synchronization service.

4.1 Physical aspects of CAN

Figure 4.1 shows the most usual architecture of a CAN network. This network is constituted by a set of nodes (the so-called *CAN nodes*) connected by means of a shared transmission medium. The internal structure of the CAN nodes is also depicted in Figure 4.1.

As in any distributed system, the processors of the CAN nodes are intended to execute some kind of common application. In order to coordinate their actions, the CAN nodes may exchange messages

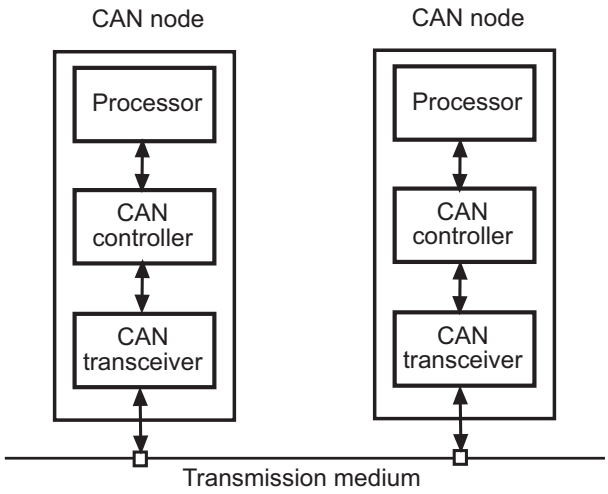


Figure 4.1: Architecture of a CAN network

through the network. The communication is performed by means of the CAN protocol, which is implemented in a hardware device called the *CAN controller*. The CAN controller provides the processor with transmission and reception services that correspond to the Logical Link Layer (Layer 2) of the OSI model [ISO93].

Every CAN controller is connected to the transmission medium by means of a specific circuit, the so-called *CAN transceiver*, which basically performs signal adaptation. CAN networks typically use a serial bus for communication. Other topologies, such as stars, are possible but rather unusual.

The bits in a CAN network are codified with NRZ (Non-Return to Zero). This causes the insertion of *stuff bits*, when required, to force edges of the transmitted signal and thus allow bit synchronization at the receivers' side. The main mechanisms of CAN are built upon the assumption that bits may take either a *dominant* or a *recessive* value. A bit value is dominant if it overwrites the existing value on the bus. A bit value is recessive if it can be overwritten by a dominant value.

In most of the current implementations of CAN, the dominant bit is represented with the logical value '0', whereas the recessive bit is represented with the logical value '1'. Due to this, it is often said that the low-level behavior of the CAN bus corresponds to a *Wired-AND*. In the rest of this document we will follow this convention.

A very important feature of CAN is the network-wide perception of the bits that it enforces. This is achieved by having the length of the bits either shortened or lengthened in order to compensate for clock tolerances and propagation delay. The idea behind this mechanism is to guarantee that all the CAN controllers of the network have a consistent view of every bit that is being broadcast. This

property is called *in-bit response*. Some interesting mechanisms are built upon this property, which are to be described in 4.3.

In order to guarantee the mentioned in-bit response, in CAN there is a relation between the bus length and the maximum bit rate achievable. Typical values are about 40m network length at 1Mbps and 1Km length for a transmission rate of 50Kbps.

4.2 CAN interface

As already mentioned, each node of a CAN network includes a controller for implementing the CAN protocol. The interface of this controller includes four primitives to handle message transmission and reception, namely Tx.Request, Tx.Confirm, Rx.Indication and Abort.Request. The first three primitives are part of the standard and are supported by any commercial CAN controller. The last primitive is not part of the CAN standard but it is supported by many available CAN controllers [Ets01].

The actual implementation of each primitive may vary from one CAN controller to another, but their function and required parameters are the following:

- The Tx.Request(msg_id, msg_data) primitive is used by the processor to request the broadcast of message *msg_id*, containing the data *msg_data*. This transmission may not take place immediately as it depends on the channel conditions.

- The Tx.Confirm(msg_id) primitive is used by the CAN controller to indicate that message *msg_id*, whose broadcast was previously requested, has been actually transmitted.

- The Rx.Indication(msg_id, msg_data) primitive is used by the CAN controller to indicate that message *msg_id* has been received, containing the data *msg_data*.

- The Abort.Request(msg_id) primitive allows the processor to request the CAN controller to abort the broadcast of the message *msg_id* (whose broadcast was requested previously by the processor itself). It is important to remark that whenever a message transmission is taking place (i.e., the bits are being transmitted on the bus), it cannot be aborted.

Commercial CAN controllers obviously provide much more functionality than these four basic primitives, both for handling transmissions and for handling receptions, and therefore their interfaces are much more complex. However, the aforementioned properties constitute the core of the CAN protocol. For this reason, in this work we will assume that these are the only services available, and will not make further assumptions about the interface of the CAN controller.

4.3 Basic mechanisms of CAN

On the basis of the guaranteed in-bit response, the CAN protocol builds some interesting mechanisms for medium access control and error control.

4.3.1 CAN arbitration for medium access control

The CAN protocol implements a prioritized access to the medium based on the message identifiers. This mechanism constitutes one of the most appealing features of CAN because of its bandwidth efficiency. Each message has a unique identifier that determines its priority with respect to the other messages. Assuming the common convention of codifying the dominant bits with the bit '0', a lower identifier implies higher priority.

Whenever the bus is idle, any CAN controller is allowed to transmit. But if more than one controller start a transmission simultaneously, then a bitwise comparison of the message identifiers is performed by each one of them. Finally, only the highest priority message is transmitted, whereas the rest of CAN controllers give up and do not try to transmit until the bus is idle again. This procedure is called the *CAN arbitration*.

Thanks to this mechanism, it is possible to bound the worst-case response time of any CAN message, as long as the traffic conditions are known in advance. I.e., as long as both the priorities and the identifiers of the messages sent by the other nodes are known in advance. Such an assumption is very common in many distributed embedded systems in which the operation conditions must be known in advance. We will hereafter denote the worst case response time of a certain message m as $wcrt_m$. The basic equations for calculating $wcrt_m$ can be found in [TBW95, DBBL07]. This analysis also shows that, given two messages m, n such that the priority of m is lower than the priority of n, then $wcrt_m < wcrt_n$.

4.3.2 Error detection and error recovery in CAN

CAN presents five error detection mechanisms to handle five types of channel error; namely bit error, stuff error, CRC error, acknowledgment error and form error [ISO93]. Whenever a CAN controller detects that a frame being transmitted suffers one of these errors, it generates a special sequence of bits (the so-called *error frame*) that corrupts the frame and makes all non-faulty CAN controllers consistently reject it, in principle.

This mechanism theoretically guarantees that every frame is either simultaneously received or consistently rejected by all the non-faulty CAN controllers. This special property of CAN is called *data*

consistency [ISO93]. Nevertheless, as it will be described later on in this section, it has been reported that some particular scenarios exist in which this property is not satisfied.

The error-recovery mechanism of CAN is based on automatic frame retransmissions. As long as the transmitter remains non-faulty, every frame corrupted by an error (and hence by the subsequent error frame) will be immediately retransmitted.

It is important to remark that, even in the presence of transient channel faults, the CAN arbitration guarantees a bounded response time for every message broadcast. A number of methods have been suggested for estimating the value of $wcrt_m$ under different traffic and error conditions. Nevertheless, the discussion of these methods goes beyond the scope of this document and is not required for understanding our work. Thus, in the following we will assume that the value of $wcrt_m$ is known for the considered fault model and traffic load. Interested readers are referred to [BBRN05, DBBL07] for a thorough discussion about determining $wcrt_m$ in the presence of faults.

4.3.3 The inconsistency scenarios of CAN

The scenarios in which data consistency is not guaranteed are related to the particular treatment of the errors in the very last bit of the End Of Frame (EOF) field. As firstly reported in [RVA$^+$98], whenever an error in the last but one bit of the EOF is only detected by a subset of the nodes, the transmitter together with the nodes belonging to this subset reject the frame, whereas the rest of nodes accept it. In the absence of node faults, the transmitter immediately retransmits the frame, thus causing that some of the nodes receive the message only once whereas the rest of nodes receive it twice. This kind of failure is called an *inconsistent message duplicate* (IMD).

As reported in [PMJ00], a second kind of inconsistency is also possible if a subset of nodes detects an error in the last but one bit of the EOF and, moreover, a second channel error prevents the transmitter from detecting the error frame. In such a situation, the erroneous frame would not be retransmitted, and then this subset of nodes would not receive a message that the other subset would have received and accepted. This failure is called an *inconsistent message omission* (IMO).

In [RVA$^+$98], it is also reported that an IMO may occur whenever a failure of the transmitter makes it impossible to retransmit a frame that has been inconsistently received. Furthermore, in [RNP03a], we reported that the inconsistency scenarios are very likely for the extensions of CAN that disable the automatic frame retransmission, such as the TTCAN protocol.

4.4 Reported dependability limitations of CAN

Despite the positive properties achieved by CAN, this protocol has a number of limitations, especially with regard to dependability, which have been reported in several publications, e.g. in [Tör95, FMD+00, APF02, SP07, PPA+09]. These limitations are briefly discussed next.

Low bit rate

This limitation is typical in shared serial data networks but it is particularly severe in CAN because of its dependency on instant bit monitoring while transmitting, a feature that in Section 4.1 was called in-bit response. Overcoming this limitation while maintaining compatibility with the standard can be achieved, for example, with different topologies (e.g., star [BARR04]) or, possibly, with segmentation (e.g., using switches).

Large and variable jitter

The arbitration mechanism implemented by CAN has the negative side effect of causing a substantial network delay jitter because, without synchronization of the transmission instants, any node will encounter all possible interference patterns from higher priority traffic when trying to transmit. This jitter can be controlled, and sometimes eliminated, using global synchronization and relative offset adjustments. The proposed techniques will be further discussed in Section 5.1.1 because they are closely related to the problem of clock synchronization.

Flexibility limitations

CAN is normally considered a highly flexible protocol. However, the arbitration mechanism is based on the message identifiers that establish the message priority and must be unique across the system. The assignment of IDs to messages has, thus, a strong impact on the timeliness of the communications and forces a system-wide fixed priority message scheduling approach.

If higher fairness is desired than achieved with fixed priorities scheduling, or to facilitate the assignment of IDs without strong consequences on the traffic timeliness, other mechanisms must be added to CAN, e.g. dynamic update of IDs or external message scheduling by means of transmission control. Moreover, all kinds of flexibility that imply dynamic changes in the message set conflict with timeliness. Combining such flexibility with timeliness requires the addition of an admission control unit that verifies all submitted changes and rejects all those that would compromise timeliness.

Limited error containment

Despite its built-in error containment mechanisms, based on error counters that can lead the network controller to bus-off state, CAN still presents several limitations in this matter. One of the limitations is that the built-in mechanisms are relatively slow to act, depending on the frequency and type of errors. Other limitation arises from the bus topology, as specified in the standard, since errors occurring in the node interfaces, bus lines or its connections, spread freely through the network causing interference with correct traffic. This may also happen with replicated buses via common-mode failures. A possible solution to this limitation consists in segmenting the network, at the physical level, e.g., using a star topology and point-to-point links.

Finally, another limitation concerns the transmission of erroneous messages, in value or timing, despite correct framing. CAN includes no protection to contain the propagation of such errors. A typical fault of this kind in the time domain is the *babbling idiot* fault in which a node remains transmitting a message more often than desired, strongly interfering with the rest of the traffic. Protection against this kind of faults may include specific hardware support, such as a bus-guardians, i.e., a device attached to a node that verifies the respective transmissions, blocking untimely ones, as well as high-layer protocols that restrict the transmission instants of nodes.

Data consistency issues

The inconsistent communication scenarios discussed in Section 4.3.3 are one of the strongest impairments to achieve high dependability over CAN. These scenarios occur due to specific protocol characteristics and they may reveal themselves both as inconsistent message omissions, i.e., some nodes receive a given message while others do not, and as inconsistent message duplicates, i.e., some nodes receive the same message several times while others receive only once. Inconsistent communication scenarios make distributed consensus in its different forms, e.g., membership, clock synchronization, consistent commitment of configuration changes, or simply the consistent perception of asynchronous events, more difficult to attain.

Limited support for fault tolerance

Safety-critical applications require very high levels of dependability (typically reliability). In order to reach these levels, fault tolerance techniques must be used. Wide support for fault tolerance functions is not a common feature in most fieldbus networks. CAN already provides advanced mechanisms which prevent some faults from causing a general system failure and even some specific CAN transceivers implement several mechanisms capable of tolerating specific permanent faults in

the communication links. However these mechanisms are not enough for safety-critical applications. Additional mechanisms are required in order to tolerate node failures and a permanent failure of the bus (i.e. to support node and bus replication).

Lack of a clock synchronization service

Distributed embedded systems can rely on a clock synchronization service in order to implement certain services that improve dependability. Unfortunately the CAN standard does not include such a clock synchronization service. Due to this, whenever a CAN-based distributed system requires a synchronized clock, it has to be provided at the application level. This is usually achieved by means of a software-implemented clock synchronization algorithm, although hardware implementations have been also proposed.

In our opinion, the lack of a clock synchronization service is one the most important limitations of CAN, mainly because it is a *hidden* requirement for solving some of the other limitations. In Section 5.1 we will show that the techniques proposed for reducing network jitter, improving error containment and supporting fault tolerance all implicitly assume the existence of such a service.

Chapter 5

Clock synchronization for dependable CAN: state of the art

It is widely accepted that the existence of a global notion of time brings significant benefits for a distributed system and, due to this, the problem of clock synchronization has received much attention during the last years, even decades. Consequently, many solutions have been already proposed, as discussed in Chapter 2. In fact, so many solutions have been already suggested that a currently generalized belief is that, regarding clock synchronization, "everything has been done".

However, when it comes to provide clock synchronization in low-cost distributed embedded systems, such as CAN-based systems, many new problems arise. Whenever a system is constrained with scarce computation and communication resources, the solutions available for large-scale distributed systems may not be directly applicable. For instance, solutions that implement fault-tolerant clock synchronization by means of complex algorithms and massive message exchanges may exceed the capacity of most low-cost processors as well as the available bandwidth of the network.

The difficulties for achieving clock synchronization in low-cost distributed embedded systems are epitomized by the case of CAN. As indicated in Section 4.4, the specification of CAN does not include a clock service, but several solutions have been proposed to implement such a service, e.g. in [GS94, Tur94, RGR98, FMD+00, RNBP03b], and [LA03]. These solutions are diverse in terms of the techniques they apply as well as in terms of the requirements they fulfill.

In this chapter we will study to what extent these solutions are adequate as a means to improve the dependability properties of CAN. For that, we will first study the techniques proposed for solving the dependability limitations of CAN, which were discussed in Section 4.4, and we will infer the

requirements they pose for the clock service. Then, these requirements will be taken as the reference for determining whether a certain solution for clock synchronization over CAN is suitable or not from a dependability perspective.

5.1 On the relevance of clock synchronization for dependable CAN

Although some of the dependability limitations of CAN may be solved with mechanisms that do not require any clock service [BPRNA06, RVA+98], a careful analysis of the available solutions shows that most of them do rely on having a very precise clock synchronization service. This is especially patent for those solutions that have been proposed for reducing network jitter, improving error containment and supporting fault tolerance, as it will be shown next.

5.1.1 Techniques for reducing network jitter

The variable network delay jitter of CAN is a consequence of its medium access control mechanism, which allows any node to start transmission whenever the bus is idle. The only way of reducing the jitter is by restricting the transmission times and the time available for retransmissions.

Some authors address this limitation by proposing static TDMA schemes over CAN [FMD+00, SP07], whereas other authors propose more flexible solutions [APF02, BB01, NNH05], which lay somewhere in between completely event-triggered communication and completely static TDMA. It is important to remark that all these solutions require the nodes to work with a global clock, what means that they actually need some mechanism for clock synchronization. Moreover, the precision achieved by this clock synchronization mechanism has a strong impact on the communication efficiency, as discussed next.

For instance, TTCAN may work without a synchronized clock (in the so-called TTCAN Level 1 mode), but high bandwidth utilization can only be achieved when a highly synchronized clock (as defined in TTCAN Level 2) is available. The clock synchronization of TTCAN Level 2 achieves a precision in the order of μs, which allows definition of longer *Basic Cycles* and tighter transmission windows, thus improving the bandwidth utilization.

In [SP07] it is highlighted that, with a bit rate of 1Mbps, an improvement of the precision from 100μs to 10μs yields an increment of the bandwidth utilization close to 40%. In [BB01], authors indicate that having a precision of 100μs would worsen performance, as it may cause either premature abortion of too many messages or an excessive number of useless retransmissions. Due to this, they recommend having a clock synchronized with a precision in the range of μs.

An interesting study about the relationship between clock synchronization and message latencies is presented in [Mah01]. This study shows that the message latencies exhibited by a TDMA scheme using a clock precision of 20μs may be 27 time lower than the latencies when using "normal" CAN.

In contrast, the solutions in [APF02, NNH05] do not need, in principle, a clock of high precision. They are based on a sort of master/slave mechanism, with a central node that periodically "polls" the nodes according to certain scheduling policy. Nevertheless, it is shown later on in this section that the mechanisms for tolerating faults of the central node will require a clock of very high precision.

5.1.2 Techniques for improving error containment

CAN includes little protection against propagation of certain faults. On the one hand, its bus topology allows the faults of certain network components (node interfaces, bus lines or connections) to spread freely and cause interference with correct traffic [BPRNA06]. On the other hand, the mechanisms to detect faulty processors or faulty CAN controllers, and isolate them from the system, are insufficient. A classical example is the *babbling idiot failure*, which is a node that broadcasts syntactically correct messages too frequently, thus causing extra interference and jeopardizing the achievement of timeliness properties [FAM$^+$03]. Another example would be a node transmitting out of its corresponding window when a TDMA scheme is used to reduce network jitter.

Very interesting solutions have been proposed to improve containment of errors caused by network components, which do not require any clock synchronization service [RVA99, BPRNA06]. Nevertheless, as long as they do not use any temporal information, they have little applicability to contain faults in the time domain, such as babbling idiots.

Error containment in the time domain can be enforced by incorporating bus guardians and high-layer protocols that limit the transmission instants, like in [FAM$^+$03] and [BB03]. But these mechanisms are efficient only when a synchronized clock is available. This is clear for the case of bus guardians, since every bus guardian needs to be highly synchronized to the node it supervises. Otherwise timing errors could not be detected, or normal situations could be interpreted as false timing errors. The precision required by these mechanisms should be in the range of μs, as we discussed previously for [BB01].

5.1.3 Mechanisms for supporting fault tolerance

Like most of current fieldbuses, CAN does not provide suitable mechanisms to support fault-tolerant applications. In particular, it does not provide any mechanisms to support node and channel replications; although various solutions have been already suggested.

In [SP07] a channel replication scheme is proposed. However, the authors of such solution claim that channel replication requires adoption of a TDMA scheme of communication, because otherwise the complexity of all possible communication patterns is overwhelming. Therefore, clock synchronization appears again as a hidden requirement to improve dependability.

Concerning node replication in CAN, current solutions are mostly built upon the assumption of high-precision clock synchronization. This is the case for the reliable broadcast service defined in [PV03]. Moreover, the master replacement mechanisms adopted by TTCAN [FMD+00] and FTT-CAN [FAF+06] require the nodes to be synchronized with a precision of a few bits (up to μs). This dependency on clock synchronization is a direct consequence of the bandwidth limitation of CAN, which discourages the adoption of completely asynchronous protocols, since they usually require transmission of a high number of messages (e.g. in [RVA+98]).

In [Mah01] it is also shown that for the case of replicated actuators, the lack of clock synchronization may be a source of inconsistencies. In that case, this problem is a direct consequence of having variable message latencies that difficult consensus.

5.2 Requirements for the clock synchronization service

It has been shown that clock synchronization is a vital service for many solutions intended to improve the dependability of CAN. Moreover, it has been shown that a precision in the order of a few μs is required by most of the techniques.

The important role played by clock synchronization turns such a service into a sort of single point of failure of the system. If clock synchronization fails (i.e. if the supplied precision exceeds certain bound) then many mechanisms that are built upon this assumption would not work properly, leading to a general failure. Therefore, the clock synchronization service not only has to be able to achieve high precision, but it must be designed in such a way that the faults of the channel and the faults of the involved nodes are tolerated.

The limited bandwidth available in CAN imposes an additional challenge to fault tolerance, since highly-distributed algorithms for clock synchronization are discouraged in favour of centralized ones. Therefore, the possibility of having Inconsistent Message Duplicates (IMD) and Inconsistent Message Omissions (IMO), as described in Section 4.3.3, becomes a severe impairment to fault tolerance that has to be taken into account.

The correctness of any fault tolerance mechanism must be properly evaluated against the faults it intends to tolerate. In most of the cases, this cannot be done by just testing or simulation since these techniques are not exhaustive, and cannot explore all possible fault scenarios. The only way to

provide absolute guarantees is through formal verification [CGP01].

When adding a new service to a system, an important aspect to be considered is the notion of *orthogonality* with respect to the rest of the system. This attribute is very desirable for two reasons. On the one hand, it allows the system designer to avoid failures caused by improper interactions between the existing mechanisms and the new service [Pro07]. On the other hand, it reduces the cost of the additional service because it requires minor changes in the rest of the system.

Our solution for clock synchronization will pay special attention to the support of fault tolerance. This constitutes one of the main contribution of OCS-CAN. Moreover, notions such as orthogonality and formal verification, which were not fully considered before in the context of clock synchronization over CAN, are specifically addressed.

5.3 Available solutions and open issues

The provision of a clock synchronization service for CAN has been studied since the emergence of this protocol. As early as in the 1st International CAN Conference, in 1994, two solutions for clock synchronization were presented [GS94, Tur94]. But these solutions mainly focused on precision and performance, and paid little attention to fault tolerance.

Later on, some other researchers envisaged the possibility of using CAN in more critical systems and suggested various algorithms which, in principle, provide fault-tolerant clock synchronization [RGR98, FMD$^+$00, LA03, APSP07, SP07]. However, the contradiction of those solutions is that they do not address fault tolerance with the desirable rigor when aiming at critical applications.

It is remarkable that, among these proposals, only the one in [RGR98] deals with a fault model that considers both node and channel faults, including the possibility of inconsistent message omissions and duplicates discussed in Section 4.3.3. Unfortunately, this clock synchronization algorithm causes high communication overhead and only achieves a precision in the order of 100μs. Thus, it does not satisfy some of the requirements posed by the techniques described in Section 5.2 and then cannot be considered a suitable solution, in general terms, for dependable CAN.

The other solutions exhibit better figures regarding overhead and precision, but they are quite incomplete with respect to fault tolerance. On the one hand, the solutions in [FMD$^+$00], [APSP07] and [SP07] propose a master/slave scheme for clock synchronization, but do not define adequate mechanisms to eliminate the single point of failure that the master represents. In [APSP07] and [SP07], master replication is not mentioned, whereas in TTCAN Level 2 it is only partially solved [FMD$^+$00]. In particular, TTCAN Level 2 defines a number of backup masters which are intended to replace the active master whenever it fails, but the algorithm to manage master replacement assumes that those

masters exhibit fail-silent behavior, and does not consider the possibility of message inconsistencies. However, these assumptions are not properly substantiated, as it would be desirable in a fault-tolerant system.

On the other hand, the solution in [LA03] relies on having several nodes that execute an agreement protocol in order to get the right time reference and spread it throughout the network. This protocol theoretically tolerates faults of several nodes. Nevertheless, it does not consider potential message inconsistencies and, more specifically, in the presence of IMOs it would not work properly.

Concerning the evaluation of the fault tolerance mechanisms, it is important to remark that the mentioned solutions have been all verified only by means of simulation or testing. None of them has been formally verified; not even the one in [RGR98]. Since simulation and testing are never exhaustive and cannot give absolute guarantees, then there is no certainty that these solutions will supply the desired level of service in every possible situation. Particularly, it is not clear that the precision values that are claimed in [LA03], [APSP07] and [SP07], which were in fact obtained in very specific experiments, would be also guaranteed in different fault scenarios.

Although it might seem that the clock synchronization service of TTCAN has been verified by model checking [LH06], it is important to remark that the model of TTCAN actually verified does not include the mechanisms for clock synchronization. More specifically, it states that "all clocks are assumed to proceed at the same rate". This circumstance contrasts with the formal verification of other time-triggered protocols, in which the clock synchronization service was the first mechanism formally verified [PSvH99].

Table 5.1 summarizes the main characteristics of the available solutions for clock synchronization over CAN. Due to their limitations, we think that at present none of the examined solutions can be considered well suited for building dependable applications over CAN. Moreover, and apart from any dependability consideration, another criticism that can be expressed about these solutions is their lack of independence (orthogonality) with respect to the rest of the system. The following dependencies are observed:

- One of the solutions requires a specific CAN controller [FMD+00], so its adoption in an existing system would require the replacement of the CAN controller of each node. This would also imply changes in the application software, mainly in the low-level communication routines.
- Some solutions require a TDMA scheme of communications [FMD+00, APSP07, SP07]. In contrast, other solutions would work only under an event-triggered scheme of communication, as they do not restrict the transmission times of the nodes, and assume that erroneous frames are automatically retransmitted [RGR98, LA03].
- Software-implemented solutions have a strong impact on the CPU schedule [RGR98, LA03,

Table 5.1: A comparison of current solutions for clock synchronization over CAN

Solution	Claimed precision	Caused overhead	Addressed fault model	Evaluation method	Comm. scheme	Hw supp.	Appl.-indep.
[GS94]	$\sim$100 μs	Low	Not addressed	Testing	Any	No	No
[Tur94]	$\sim$10 μs	Low	Not addressed	Testing	Any	Yes	No
[RGR98]	$\sim$100 μs	High	Node & channel faults	Simulation & Testing	E-T	No	No
[LA03]	$\sim$4 μs	Medium	Only node faults	Testing	E-T	No	No
[FMD$^+$00]	$\sim$10 μs	Low	Node & (consist.) channel faults	Testing	T-T	Yes	No
[APSP07]	$\sim$2 μs	High	Not addressed	Testing	T-T	No	No
[SP07]	$\sim$2 μs	High	Channel faults	Testing	T-T	No	No

APSP07, SP07]. Clock synchronization is a demanding task, which has to be executed with a very high priority in order to get high precision. Moreover, whenever fault tolerance is to be provided with massive message exchanges, like in [RGR98] and [LA03], the computation requirements raise significantly.

Chapter 6

The Orthogonal Clock Subsystem for CAN

In Chapter 4, we reviewed the main limitations of CAN concerning dependability and we indicated that one of the most important limitations is the lack of a clock synchronization service. In Chapter 5 we showed that the provision of a precise clock synchronization service is required for implementing many solutions for dependability on CAN, and discussed which requirements these solutions impose upon the clock service.

In Chapter 5, we also presented the available solutions for implementing clock synchronization over CAN, and showed that none of them really provides a service fulfilling the stated requirements. Therefore, there is room for a new solution. This chapter discusses the design and implementation of our proposal for providing clock synchronization over CAN: the Orthogonal Clock Subsystem for CAN (OCS-CAN). This novel solution is intended to fulfilling the strict requirements imposed by the techniques for dependable CAN.

6.1 Preliminary remarks about our proposal

OCS-CAN is a proposal for clock synchronization over CAN that specifically addresses the requirements discussed in Section 5.2. OCS-CAN attempts to balance the advantages and drawbacks of the previous solutions, with the aim of providing a service more suitable for dependable applications. Another important premise of our solution is *simplicity*; not only because simplicity reduces the cost

of the solution, but also because it simplifies the provision of fault tolerance as well as the evaluation of the system.

In this sense, we have to stress that the novelty of the design is not one of the main strengths of our proposal. Most of the techniques that are adopted in the design of OCS-CAN are not innovative. However, what is innovative in our solution is the thorough evaluation of the correctness of the fault tolerance mechanism, together with the formal assessment of the effect that faults may have on the precision provided by OCS-CAN.

The assessment of OCS-CAN will be performed analytically and also by means of model checking; these evaluations will be discussed in Chapter 7 and in Chapter 9, respectively. The aim of our assessment will be to quantify the relationship between the faults of the system and the achievable precision. We will be able to characterize the *graceful degradation* exhibited by our solution in the presence of faults. In our evaluation of OCS-CAN by means of model checking, we had to overcome some serious limitations of the timed automata formalism. This is a very important contribution of this work, that goes beyond our initial goal, and will be discussed in Chapter 8.

OCS-CAN has been also developed and evaluated in the laboratory. However, this implementation is meant as a proof of concept, which shows that our design can be implemented as a relatively simple digital system, and in particular over a medium-range Field Programmable Gate Array (FPGA).

6.2 Properties of the orthogonal clock subsystem

In accordance to the requirements discussed in Section 5.2, OCS-CAN pursues four properties: high precision, low overhead, fault tolerance and cost effectiveness. The techniques that are adopted to achieve those attributes are discussed here.

6.2.1 High precision

The precision achieved by a clock synchronization algorithm basically depends on the latency of the mechanism for timestamping the resynchronization event of the first phase. Whenever this mechanism is implemented in hardware, the precision is in the order of a few μs, whereas in software the precision is only of about hundreds of μs ([Kop97], Table 3.2).

To improve precision, OCS-CAN adopts the hardware mechanism suggested by Turski [Tur94] to timestamp incoming frames. This mechanism relies on a modified CAN controller that signals the sampling point (shown in Figure 6.1) of a very particular bit of every frame: the Start of Frame (SOF) bit. By applying this mechanism, the offset between nodes can be calculated very accurately, because

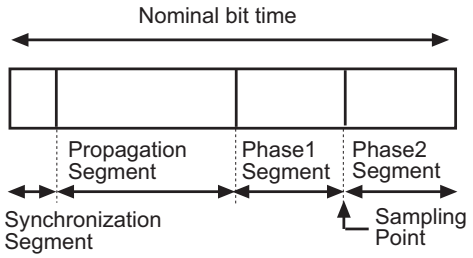


Figure 6.1: Structure of a CAN bit and location of the sampling point

timestamps are free from the uncertainty caused by network jitter as well as by message processing jitter.

Unfortunately, at present there are no available CAN controllers that signal the sampling point of the SOF bit. Although in the past this feature was supported by at least one CAN controller [Tur94], it seems that manufacturers do not include this service any longer. This forces us to develop a modified CAN controller, especially tailored to signal this event.

6.2.2 Low overhead

In order to reduce overhead, OCS-CAN uses a master/slave algorithm. This means that, in principle, only one node (the master) is allowed to spread its time view and the rest of nodes (the slaves) synchronize to it. To spread its time view, the master periodically broadcasts a synchronization message, which is called the Time Message (TM). Therefore, the timestamp mechanism described above has to be applied to the SOF bit of this particular message.

Moreover, and it order to further reduce the number of required messages, the master piggybacks its timestamp within the data field of the TM. This mechanism, which is illustrated in Figure 6.2, is very similar to the one used with the *reference message* of TTCAN Level 2 [FMD$^+$00]. Note that in this manner, the Phase 1 and Phase 2 of the clock synchronization algorithm are merged into a single phase. The resultant communication scheme when no channel error exist is shown in Figure 6.3. In case of channel errors, every subsequent retransmission of the TM will contain the timestamp corresponding to the new SOF bit.

6.2.3 Fault tolerance

Concerning fault tolerance, OCS-CAN presents several advantages with respect to previous solutions. Our strategy for fault tolerance is based on three points:

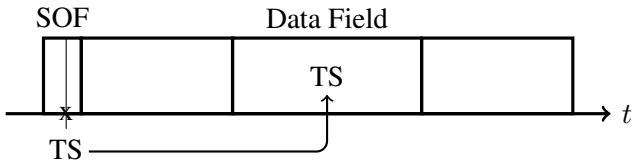


Figure 6.2: For each TM broadcast, a timestamp is taken at the sampling point of the Start Of Frame bit, and it is written in the data field

1. Restriction of the failure semantics of the nodes to *crash failure semantics*. The design of the mechanisms to ensure this property has not been addressed in this work, but in the literature about fault tolerance it is possible to find several techniques for restricting the failure semantics of a computer system. For instance, with a duplication and comparison scheme as the one described in [PPMJ99], and assuming that nodes are simple enough to be free from design errors, it is theoretically possible to ensure crash failure semantics. However, the adaptation of these techniques to the particular case of OCS-CAN has not been investigated yet.

2. Use of master replication in order to eliminate the single point of failure that the master represents. This is thoroughly discussed in Section 6.3.4.

3. Use of formal verification in order to assess the correctness of the fault tolerance mechanisms. This formal verification shall consider a wide fault model, explicitly including the possibility of message inconsistencies. The formal verification of OCS-CAN is performed by means of model checking, and it will be discussed in detail in Chapter 9.

Permanent failures of the channel, such as bus partition or stuck-at-dominant, are not addressed by OCS-CAN. Nevertheless, they could be tolerated with mechanisms such as the one defined in [RVA99], which are fully compatible with our architecture.

6.2.4 Cost issues

The cost of our solution is inevitably higher than the cost of a CAN system without our clock synchronization service. However, such an increment is justified by the strict requirements that our solution fulfills. First, addition of extra hardware is required in order to improve the precision of the timestamp, and second, extra hardware has to be also incorporated to restrict the failure semantics of the nodes. Nevertheless, we can still reduce the cost associated to the new service thanks to the notion of *orthogonality*.

Our proposal achieves orthogonality by implementing the clock synchronization service into a set

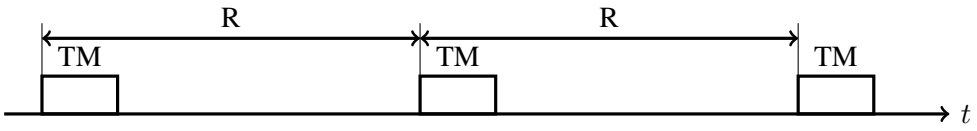


Figure 6.3: Transmission pattern of the master in the absence of faults

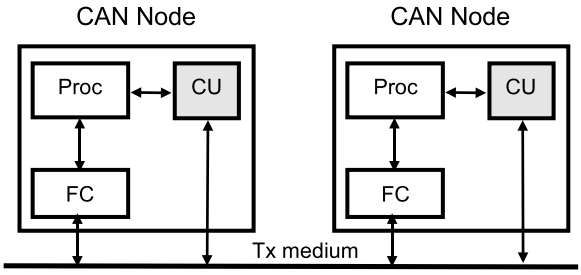


Figure 6.4: Architecture of OCS-CAN

of independent modules. As depicted in Figure 6.4, OCS-CAN is made up of a set of hardware components, named *clock units* (CU), such that a clock unit is attached to every node in the system. Every clock unit can behave either as a master or as a slave.

In order to send (only for the case of master clock units) and receive TMs, every clock unit is connected to the same CAN bus as the processor. Nevertheless, the clock unit does not use the same CAN fieldbus controller (FC in Fig. 6.4), since each clock unit is provided with its own CAN controller and CAN transceiver. This is described in more detail in Section 6.3.1, when discussing the internal structure of the clock unit.

In this manner, OCS-CAN does not require replacement of previous CAN controllers nor reprogramming of existing applications. Moreover, the processor is released from executing the clock synchronization algorithm so that the impact on the application software, as well as on the processor scheduling, is drastically reduced. Furthermore, this independence simplifies the design of the clock unit, and more importantly, makes the results of the formal verification not depend on the characteristics of the system to which OCS-CAN is incorporated. Our aim is that system designers be able to add our clock service very easily to any CAN architecture, knowing that the service is reliable and that no side-effects exist.

6.3 Description of the architecture

Once the main mechanisms of OCS-CAN have been presented, we can delve deeper into the description of its architecture.

6.3.1 Internal structure of the clock unit

The internal structure of the clock unit, without considering the duplication and comparison mechanism, is depicted in Figure 6.5. It is made up of the following blocks: the Virtual Clock module, the Synchronization Module, the Timestamp Manager and the Enhanced CAN Controller. It is important to remark that the only difference between a master and a slave clock unit is the behavior of the Synchronization Module. The rest of modules remain unchanged.

The Virtual Clock module provides the current value of time (`vc`) to the processor and to the modules that need it. This module also performs the third phase of the clock synchronization algorithm, the clock adjustment procedure, upon request via the `Sync` primitive.

The Synchronization Module (SynM) is intended to perform the high-level functions of our synchronization algorithm, including the mechanism for managing master redundancy to be discussed in Section 6.3.4. This module also calculates the offset with respect to the master time reference when appropriate, and signals through the `Sync` primitive the moment to recalculate the clock adjustment.

The Timestamp Manager (TSM) deals with the low-level functions of the clock synchronization algorithm, which mainly concern the timestamping of every outgoing and incoming TM. This is better explained later on in this section, after introducing the function of the Enhanced CAN Controller.

The Enhanced CAN controller (EnCAN) provides the usual transmission and reception functions of any CAN controller, plus two additional services: the signaling of the sampling point of every SOF bit and the possibility of overwriting the data field while a CAN frame is being transmitted. Both services allow TSM to implement a very accurate timestamp of every TM, as discussed next.

6.3.2 Timestamp mechanism

Timestamps are obtained by TSM, in coordination with its corresponding EnCAN, by means of a simple mechanism: whenever EnCAN signals the sampling point of a SOF bit, TSM takes a sample of `vc`. This is called the *local* timestamp of the message and is denoted with T^i_{loc}.

Note that each SOF signaling may be caused either by a message which is being transmitted by EnCAN or by a message which is being received. In each case, the local timestamp will be treated

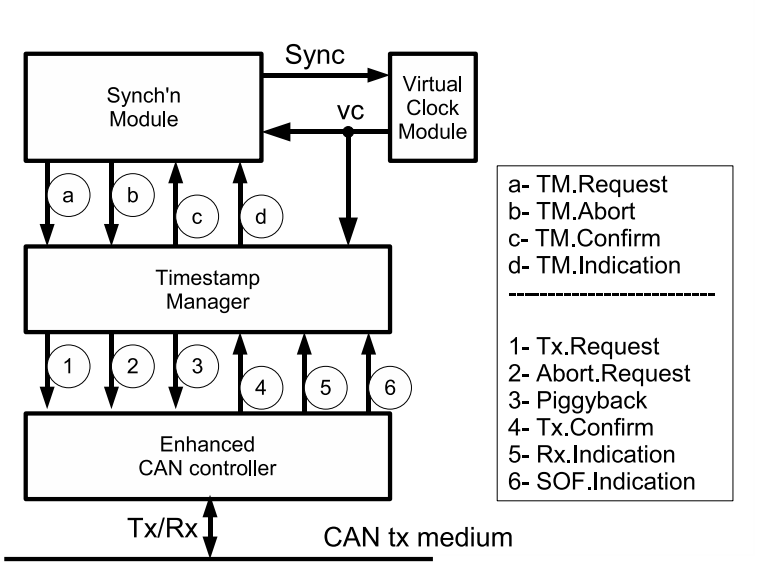


Figure 6.5: Block diagram of the clock unit, with the interface between blocks

differently. If EnCAN is the transmitter of the message then TSM uses the `Piggyback` primitive to overwrite the data field of the message with the value of the local timestamp. In contrast, if EnCAN is receiving the message then TSM waits until it has been entirely received, which is notified with the `Rx.Indication` primitive, and then checks the message identifier. If the received message is a TM then TSM extracts the (piggybacked) *remote* timestamp (denoted as T^i_{rem}) and saves it locally. After that, the reception of the TM is notified to SynM (with a `TM.Indication`) and both the local and the remote timestamps are passed on to that module.

6.3.3 Clock adjustment

For every resynchronization round i, the current offset (Φ^i) is calculated as the difference between the remote and the local timestamps: $\Phi^i = T^i_{rem} - T^i_{loc}$. The aim of the clock adjustment is to compensate the current offset and try to minimize the offset for the next resynchronization round.

Nevertheless, the clock adjustment should also consider the adjustment done in previous rounds, and keep it. Note that a value $\Phi^i = 0$ does not mean that the clock adjustment must be stopped, but it means that the *same* clock adjustment must be performed during the next round. We define the *accumulated offset* (θ^i) to account for this factor. For every resynchronization round i, it is calculated as $\theta^i = \theta^{i-1} + \Phi^i$.

In OCS-CAN, clock adjustment is carried out by means of *amortization*, what means that the

correction term is not applied in a discrete step, but it is progressively applied throughout all the resynchronization round. Such smooth adjustment is more suitable from the application's point of view and does not cause any loss of precision [SC90].

The value of the virtual clock is supplied by a counter, which is fed by a frequency divider that provides a clock of frequency $\frac{f_{osc}}{\mathrm{N}}$. The way to apply the correction term is by speeding up or slowing down the output of the frequency divider. In our system, this is performed by substracting/adding 1 unit from/to N.

The proportion of ticks that are affected by a change of N depends on the absolute value of the correction term: a large correction term requires more frequent changes than a small one. Our mechanism forces 1 of every k ticks to be shortened/lengthened. The value of k is recalculated in every resynchronization round i as follows.

$$k^i = \frac{T^i_{rem} - T^{i-1}_{rem}}{|\theta^i + \Phi^i| \cdot \mathrm{N}} \tag{6.1}$$

The sign of $\theta^{i-1} + \Phi^i$ indicates whether the length of the virtual clock ticks must be shortened of lengthened. If the sign is positive then $\frac{1}{k}$ of the virtual clock ticks will have length $\mathrm{N} - 1$ ticks of the local oscillator, whereas if it is negative then $\frac{1}{k}$ of the ticks will have length $\mathrm{N} + 1$.

6.3.4 Algorithm for managing master redundancy

Although the failure semantics of the clock units is restricted, failures of the master may still occur, so the system must be ready to deal with such situations. As already indicated in Section 6.2.3, OCS-CAN defines a number of backup masters, one of which shall replace the master whenever it fails.

The mechanism for master replacement assumes that masters are organized hierarchically. The priority of a master is defined with two parameters. First, the identifier of the TM codifies the priority of the master that broadcasts it. Following the common convention in CAN, a lower identifier implies higher priority. Second, in every resynchronization round, every master releases its TM at a different instant. This time instant is called the *broadcast instant*, and it is defined, for master m in the resynchronization round k, as $\mathrm{T}_{rls_m} = k\mathrm{R} + \Delta_m$, where R is the resynchronization period (the same for all masters) and Δ_m is a small delay, whose length is inversely proportional to the priority of the master.

This mechanism can be formalized as follows. The set of masters with higher priority than a given master m is denoted as $hp(m)$. Then, for every pair of masters n, m in an OCS-CAN system, the following conditions are satisfied:

- If $n \in hp(m)$ then the identifier of the TM broadcast by master n is lower than the identifier of

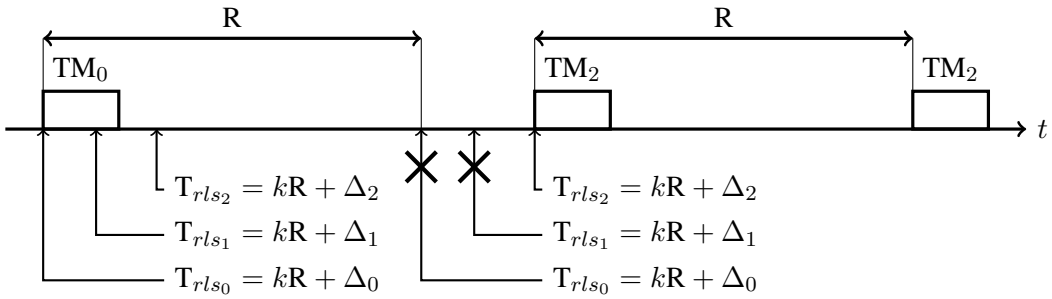


Figure 6.6: Master replacement upon failure of two masters

the TM broadcast by master m.

- If $n \in hp(m)$ then $\Delta_n < \Delta_m$.

This assignment of identifiers, combined with the definition of the broadcast instants, guarantees that in a synchronization round, a master may successfully broadcast its TM before a master of higher priority only if the latter is faulty. This is depicted in Figure 6.6, for the case of three masters. Notice that, although Δ_m is by definition much shorter than the period (R), this difference is not visible in Figure 6.6. This has been done for the sake of clarity, since the purpose of the graph is only to show how the master replacement takes place.

This hierarchical approach significantly simplifies the algorithm to be executed by the SynM of a master clock unit. This algorithm is shown in Figure 6.7 for master m. Every master (in fact, the SynM of every master) implements the same algorithm, being the only difference the value of Δ_m and the value of the TM identifier.

The initial state of SynM is `Idle1`. While being in this state, three events may occur.

1. A TM of lower priority can be received. This is expressed with the condition TM.Ind$(n) \wedge\ n \notin hp\ (m)$. Whenever this happens, the received TM is just ignored and SynM remains in the same state.

2. A TM of higher priority may be received. This is expressed with the condition TM.Ind$(n) \wedge n \in hp\ (m)$, where n is the identifier of the TM just received. In this case, SynM notifies the VC module through the Sync(n, m) primitive that the procedure for clock adjustment has to be executed, and goes to state `Idle2`.

3. The master may reach its broadcast instant, which is expressed with the condition $vc_m(t) = kR + \Delta_m$. Whenever this happens, SynM requests the broadcast of its TM, via the TM.Req(m) primitive, and steps into state `Queue`.

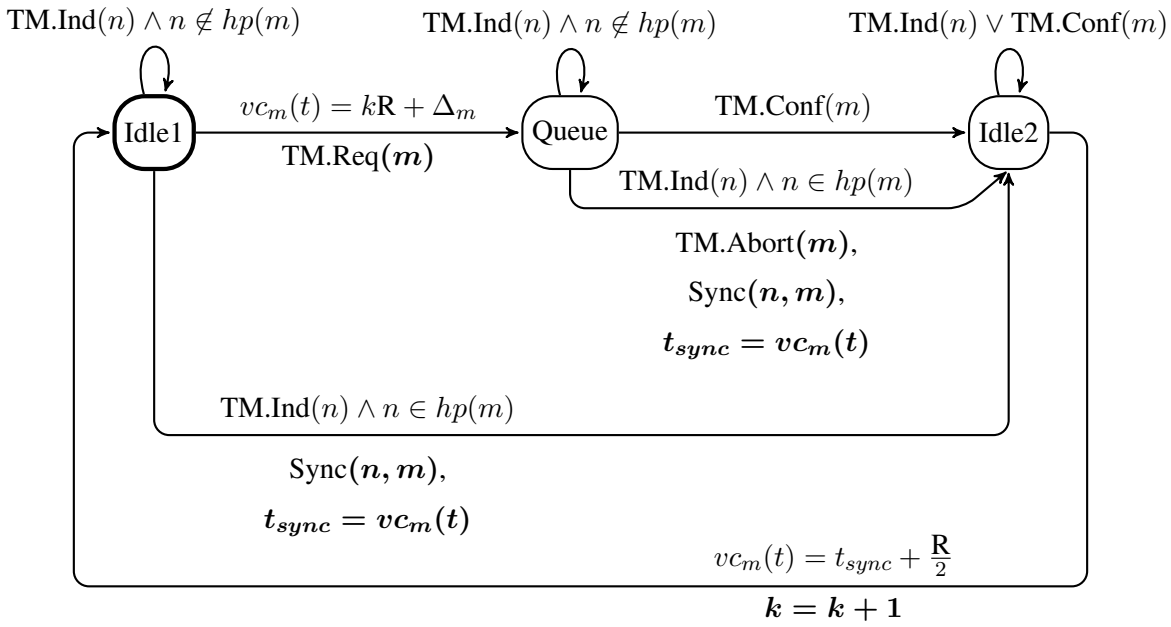


Figure 6.7: Algorithm executed by the SynM of master m

In state `Queue`, three events are possible.

1. A TM of lower priority can be received. This is expressed again with the condition TM.Ind$(n) \wedge n \notin hp\ (m)$. Whenever this happens, the received TM is ignored.

2. A TM of higher priority may be received. This is expressed also with the condition TM.Ind$(n) \wedge n \in hp\ (m)$, where n is the identifier of the TM just received. In this case, the previously requested broadcast is aborted (with the TM.Abort(m) primitive) and SynM notifies the VC module (through the Sync(n, m) primitive) that the procedure for clock adjustment has to be executed.

3. The broadcast of the TM may be successful. This would be indicated by TSM with the TM.Conf(m) primitive. In this case, SynM steps into state `Idle2` without performing any particular action.

Regarding the TM.Abort(m) primitive, it is important to remark that under some particular circumstances it may not be successful. More specifically, due to the inherent latency in the processing of any received message, SynM issues the TM.Abort(m) with a short delay after the reception of the TM. It is possible that during this delay, EnCAN has gained access to the bus and is actually transmitting the TM that should be aborted. In such a case, the abortion could not take place. As any other

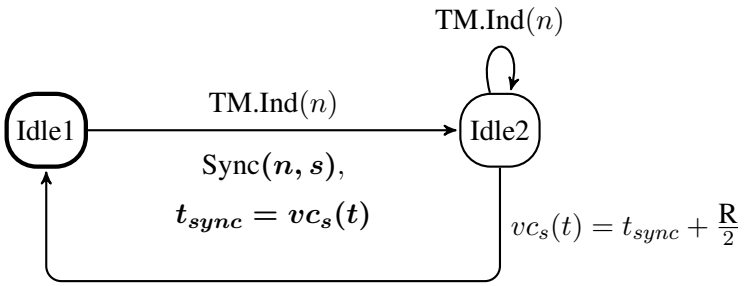


Figure 6.8: Algorithm executed by the SynM of slave s

CAN controller, EnCAN can only abort those messages that are waiting for transmission, but not the one that it is transmitting.

An unsuccessful abortion of the TM would cause the reception of two different TM within the same resynchronization round. In order to prevent the second TM from causing another resynchronization, the master waits for $\frac{R}{2}$ time units after every synchronization (in state `Idle2`), before being available for synchronizing again. Meanwhile, every received TM is ignored, regardless of its priority. If the master was not able to timely abort its previous broadcast, the TM.Conf(m) primitive is also ignored. In Section 7.4 it will be shown that this technique is also useful to avoid problems with duplicates of the TM caused by channel faults.

Figure 6.8 depicts the algorithm executed by the SynM of the slave s. The behavior of a slave is very simple: it synchronizes to the first TM received in every round, regardless of the clock unit that generated the TM. In order to avoid double resynchronizations caused by TM duplicates, only TMs that are at least $\frac{R}{2}$ time units apart are considered.

6.4 Prototype and testing of OCS-CAN

The prototyping of OCS-CAN has been addressed as a *Projecte Final de Carrera* of the degree of *Enginyeria Tècnica Industrial, especialitat Electrònica Industrial* at the UIB [Fer08]. The first aim of this project was to fully develop the internal structure of the modules that constitute the clock unit, and to prove that the resulting circuit can be implemented at a reasonable cost with a *Programmable Logic Device* (PLD).

For illustration purposes, Figure 6.9 shows the block diagram that corresponds to TSM, as it appeared in [Fer08]. This module includes a number of submodules (Priority check, Rx_ref_msg, Tx_ref_msg and Timestamp_module) and a more complex interface than shown in Figure 6.5. However, these low-level details of the implementation will not be discussed here.

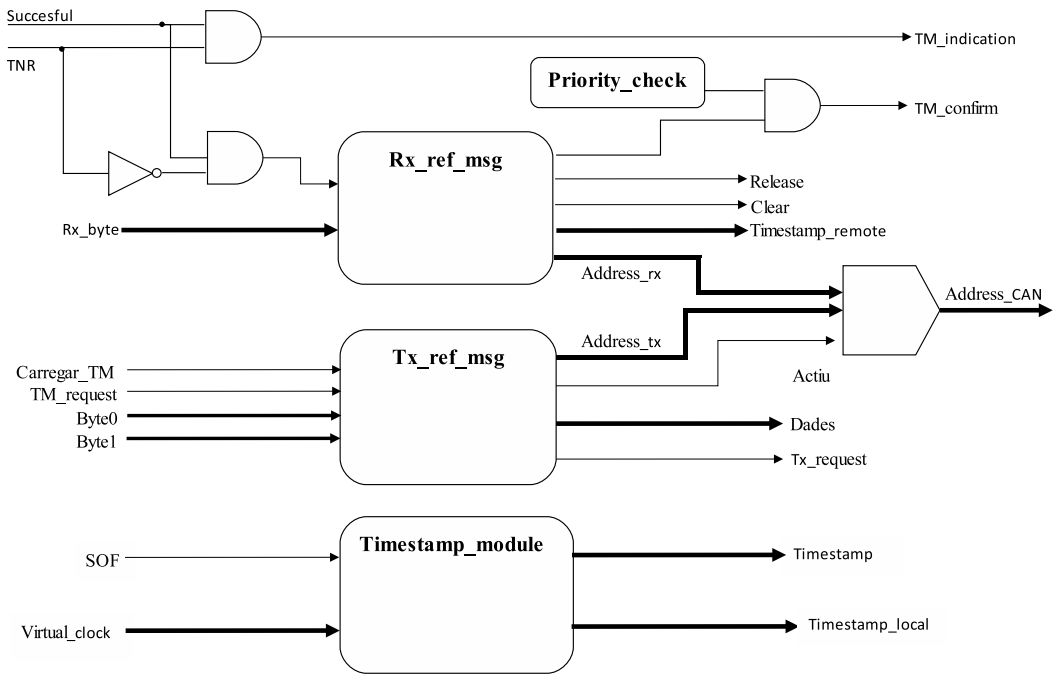


Figure 6.9: Block diagram of the Timestamp Manager (TSM)

The prototype of the clock unit was finally implemented on an EP20K200EFC484-2X, which is an FPGA belonging to the family APEX20KE from Altera [Alt]. This device can be currently considered a low-range PLD, what confirms that the hardware requirements of OCS-CAN are not hard to meet.

The second aim of the prototyping of OCS-CAN was to measure the precision achieved by the clock adjustment procedure, and determine whether this precision satisfies the requirements discussed in Section 5.2. To obtain this measure, we set up a simple system, made up of only one master and two slaves, connected with a CAN network at 500Kbps.

In our setup, every clock unit is clocked with a local oscillator of frequency 25MHz, which feeds the frequency divider that generates the ticks of the virtual clock (see Section 6.3.3). For testing purposes, the output of this frequency divider is connected to an output pin of the FPGA, which we will call `vc_tick` hereafter. The frequency divider works with a default value of $N = 50$ so that the period of the virtual clock (and hence of `vc_tick`) is 2 μs.

It is important to remark that having a virtual clock of period 2 μs (a relatively slow clock) limits the precision achievable in our experiment. With this clock granularity, even simultaneous events can be timestamped with an error of 2 μs, which implies that the timestamps already include some error that cannot be corrected. For this reason, our has to use a short resynchronization period (R).

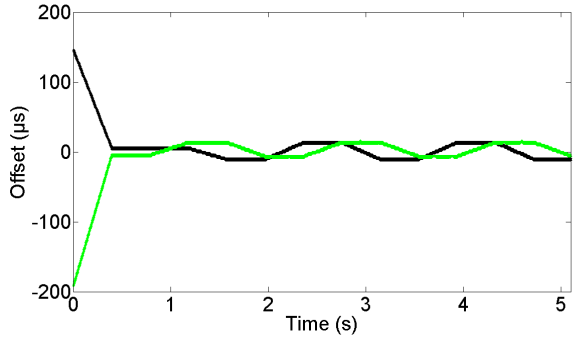


Figure 6.10: Offset measured for two slave clock units

The offset among the virtual clocks is measured by periodically sampling the outputs `vc_tick` of the master and the two slaves. The samples are taken with a digital oscilloscope Yokogawa DL7440, at a frequency of 2 Msamples/s. This oscilloscope has four channels, three of which are used to monitor the pin `vc_tick` of each clock unit. This guarantees that the samples are taken simultaneously. After that, the recorded samples are processed off-line with Matlab in order to obtain the value of each virtual clocks, and compare it with the value of the master's virtual clock.

Figure 6.10 depicts the results that were obtained with a resynchronization period R $= 0.3$ s. The horizontal axis indicates time, measured in seconds, whereas the vertical axis indicates the offset with respect to the master's virtual clock, measured in ticks of the master. Initially, the slave virtual clocks have an offset of 60 and -100 ticks (120 μs and -200 μs, respectively). Both initial offsets are generated artificially in order to check the convergence of the clock adjustment.

As Figure 6.10 shows, the offset is corrected in only one resynchronization round. After that, the virtual clocks of the slaves keep oscillating around the central value. The maximum offset with respect to the master is ± 5 ticks, which means that the system achieves a precision of 10 ticks (20 μs).

As indicated in Section 5.2, this value stays within the range of acceptable values for the design of dependable CAN. Although it is clear that more experiments are required in order to further investigate the precision achieved by our prototype, we consider that the results obtained so far show that the proposed architecture can be implemented with relatively economic hardware.

Nevertheless, it is important to remark that, when aiming at dependable systems, testing is not enough. Tests can show the existence of design errors, and can also help to assess the average behavior of the system, but tests cannot prove that the system behaves as expected in every possible situation. In order to provide such guarantees a formal approach is required. This will be discussed in the next chapter, in which we present the analytical assessment of OCS-CAN.

Chapter 7

Analytical assessment of the precision guaranteed by OCS-CAN

This chapter addresses the analysis of the precision achieved by OCS-CAN under diverse fault conditions. This analysis considers the clock units independently from the rest of the distributed embedded system. This is possible thanks to the orthogonality of the clock subsystem.

7.1 Basic definitions and notation

In this section, we present the equations that model the behavior of the virtual clocks, and use them to give a formal definition of the internal clock synchronization property. Two basic results to be used in the subsequent analysis of OCS-CAN are also presented.

7.1.1 Characterization of the virtual clock

In this work we assume the Newtonian concept of time, as it is usually done in computer systems. This notion of time, which is also referred to as *real time*, is represented by the set of positive real numbers, $\mathbb{R}^+$. Any time instant belonging to this external dimension will be denoted by t.

The virtual clock is a counter that uses a local oscillator for measuring real time as accurately as possible. The signal generated by the local oscillator goes through a frequency divider, which divides by a constant number N. The virtual clock therefore increases in one unit every $\frac{N}{f_{osc}}$ seconds. For

any clock unit a, we denote the value of its virtual clock (measured in seconds) as $vc_a(t) \in \mathbb{R}^+$.

Note that the virtual clock can only take discrete values because it increases stepwise (tick by tick). Nevertheless, to simplify our analysis we assume that $vc_a(t)$ is a continuous, differentiable and monotonically increasing function, such that $\dot{vc}_a(t) > 0$ for every time instant t. Since the increment caused by a tick is very little, and ticks are actually very close to each other, this approximation does not cause a significant error. It is indeed a common assumption when addressing the study of clock synchronization [Sch87, FC95, PSvH99].

According to this definition, an *ideal* virtual clock is the one that perfectly maps real time, and thus satisfies $\dot{vc}_a(t) = 1$ for any time instant t. However, actual virtual clocks never exhibit such an ideal behavior, mainly because local oscillators inevitably deviate from their nominal frequency.

For any clock unit a, the rate of its virtual clock is defined as $\dot{vc}_a(t) = 1 + \rho_a(t)$, where $\rho_a(t) \in \mathbb{R}$ is called the *drift* of virtual clock a. The drift of a non-faulty virtual clock a is always bounded by a value ρ_a^{max}, which is called its *maximum drift*, such that $|\rho_a(t)| \leq \rho_a^{max} \ \ \forall\, t \in \mathbb{R}^+$.

The value of ρ_a^{max} is much lower than 1, usually in the order of 10^{-4} to 10^{-6}, what implies that virtual clocks approximately evolve at the pace of real time.

The drift of a local oscillator (and thus of a virtual clock) is a combination of two components: the *stochastic* drift and the *systematic* drift. The systematic drift, which is caused by manufacturing imperfections and aging, but also by environmental conditions such as temperature variations, is two or three orders of magnitude higher than the stochastic one. Due to this, the stochastic drift is usually neglected.

Since the systematic drift changes very smoothly, it is common to assume that $\dot{\rho}_m(t) \simeq 0$. This assumption is called *long-term stability* and implies that for short periods of time, and as long as the rate is not changed by any clock synchronization action, $\dot{vc}_a(t)$ is constant and thus the function $vc_a(t)$ can be satisfactorily approximated by a linear function.

7.1.2 Offset, consonance and precision

The consistency in the perception of time among a set of clock units is usually studied by means of a measure called the *offset*, which is defined as follows.

Definition 4 (Offset)**.** *Let $a, b \in A$ be two clock units. The* offset *between clock unit a and clock unit b at time t is $\Phi_{ab}(t) = vc_a(t) - vc_b(t)$.*

In order to perform coordinated actions in a distributed system, the most desirable situation would be to have an offset equal to 0 between every pair of clock units, as it would allow them to easily make

consistent decisions on the basis of time. Unfortunately, such a desirable situation is not possible in real systems, since every virtual clock evolves at a slightly different rate, because of its specific drift, and therefore the values of the virtual clocks tend to diverge.

The speed at which two virtual clocks diverge depends on a measure called the *consonance*, which indicates their relative rate difference.

Definition 5 (Consonance)**.** *Let $a, b \in A$ be two clock units. The* consonance *between clock unit a and clock unit b at time t is $\gamma_{ab}(t) = \dot{vc}_a(t) - \dot{vc}_b(t) = \rho_a(t) - \rho_b(t)$.*

When $vc_a(t)$ and $vc_b(t)$ are approximated by linear functions, the offset and the consonance are related as follows $\Phi_{ab}(t) = \Phi_{ab}(t_0) + (t - t_0)\gamma_{ab}(t_0)$ or, expressed in absolute value, $|\Phi_{ab}(t) - \Phi_{ab}(t_0)| = (t - t_0)|\gamma_{ab}(t_0)|$. This means that the tendency of the offset is to increase (in absolute value) as time passes by; except for the ideal case when $\gamma_{ab}(t_0) = 0$.

Therefore, as long as virtual clocks are allowed to run free, it is not possible to rely on their values in order to have a consistent perception of time. As indicated in Section 2.2, this can be solved with a *clock synchronization algorithm* that ensures the property known as *internal clock synchronization*. This property is formalized as follows.

Definition 6. *[Internal clock synchronization] Let A be a set of clock units. Set A is* internally synchronized with precision Π *if there exists a constant $\Pi \in \mathbb{R}^+$, which is called the* precision*, such that for every pair $a, b \in A$ of non-faulty clock units, $|\Phi_{ab}(t)| \leq \Pi$, $\forall\, t \in \mathbb{R}^+$.*

For short, we will say hereafter that a set of clock units fulfilling this property is Π-*synchronized*.

7.1.3 Two basic results on virtual clocks

Thanks to the linearity properties assumed for $vc_a(t)$, the following two lemmas can be derived.

Lemma 1. *Let $a \in A$ be a non-faulty clock unit, and t_1, $t_2 \in \mathbb{R}^+$ be two time instants such that $t_1 < t_2$. If there exist four values $\tau_l, \tau_u, v_l, v_u \in \mathbb{R}^+$ such that $\tau_l \leq t_2 - t_1 \leq \tau_u$, and $v_l \leq \dot{vc}_a(t) \leq v_u$ $\forall\, t \in [t_1, t_2]$, then*

$$\tau_l \cdot v_l \leq vc_a(t_2) - vc_a(t_1) \leq \tau_u \cdot v_u.$$

Proof. Assuming that $vc_a(t)$ is a linear function, we know that $vc_a(t_2) - vc_a(t_1) = (t_2 - t_1)\dot{vc}_a(t_1)$.

Then, by hypotheses,

$$\tau_l \cdot v_l \leq (t_2 - t_1) \cdot v_l \leq (t_2 - t_1)\dot{vc}_a(t_1) \leq (t_2 - t_1) \cdot v_u \leq \tau_u \cdot v_u.$$

□

Lemma 2. *Let $a \in A$ be a non-faulty clock unit, and t_1, $t_2 \in \mathbb{R}^+$ be two time instants such that $t_1 < t_2$. If there exist four values $x_l, x_u, v_l, v_u \in \mathbb{R}^+$ such that $x_l \leq vc_a(t_2) - vc_a(t1) \leq x_u$, and $v_l \leq \dot{vc}_a(t) \leq v_u \; \forall \; t \in [t_1, t_2]$, then*

$$\frac{x_l}{v_u} \leq t_2 - t_1 \leq \frac{x_u}{v_l}.$$

Proof. Assuming that $vc_a(t)$ is a linear function, we know that $vc_a(t_2) - vc_a(t_1) = (t_2 - t_1)\dot{vc}_a(t_1)$ and therefore

$$\frac{vc_a(t_2) - vc_a(t_1)}{\dot{vc}_a(t_1)} = t_2 - t_1 \ .$$

Then, by hypotheses,

$$\frac{x_l}{v_u} \leq \frac{x_l}{\dot{vc}_a(t_1)} \leq t_2 - t_1 \leq \frac{x_u}{\dot{vc}_a(t_1)} \leq \frac{x_u}{v_l} \ .$$

□

Both lemmas are extensively used in the analysis that is carried out in the following sections.

7.2 The clock synchronization algorithm of OCS-CAN

In this section we give the equations that model the operation of the clock synchronization algorithm of OCS-CAN, assuming that faults may not exist. This analysis will be complemented in Section 7.4 and Section 7.5 with the assessment of the fault tolerance mechanisms of OCS-CAN.

7.2.1 Modeling clock adjustment

Let a be a slave clock unit, and let t^i_{sync} be the value of real time at the sampling point of the SOF bit of the TM that the master broadcasts in round i. Then, after the complete reception of the TM, clock

unit a has two timestamps of that instant: the one sent by the master, $vc_m(t^i_{sync})$, and the one it has taken, $vc_a(t^i_{sync})$. The first value is called the *remote timestamp* of the TM and is denoted by T^i_{rem}, whereas the second value is called the *local timestamp* of the TM and is denoted by T^i_{loc}. Both values can be used in order to estimate the offset with respect to the master, since $\hat{\Phi}_{ma}(t^i_{sync}) = T^i_{rem} - T^i_{loc}$.

Due to the hardware-implemented timestamp mechanism of OCS-CAN, explained in Section 6.3.2, the obtained estimation is very accurate and satisfies the following property

$$|\Phi_{ma}(t^i_{sync}) - \hat{\Phi}_{ma}(t^i_{sync})| \leq \epsilon_0 \ .$$

Furthermore, every clock unit can also calculate its consonance with respect to the master. For this calculation, the timestamps from two consecutive synchronization rounds must be used as follows

$$\hat{\gamma}_{ma}(t^i_{sync}) = \frac{T^i_{loc} - T^{i-1}_{rem}}{T^i_{rem} - T^{i-1}_{rem}} \ .$$

Thanks to the accuracy of the timestamps, such an estimate of the consonance approximates the real consonance very closely. Thus, the property $|\gamma_{ma}(t^i_{sync}) - \hat{\gamma}_{ma}(t^i_{sync})| \leq \gamma_0$ is fulfilled.

Once a slave clock unit has calculated both its offset and its consonance with respect to the master, the clock adjustment can be easily carried out. It is divided into two steps: the offset adjustment and the drift adjustment. The offset adjustment can be performed by means of an assignment to the virtual clock, as follows

$$vc_a(t^i_{sync}) := vc_a(t^i_{sync}) + \hat{\Phi}_{ma}(t^i_{sync}) \ .$$

Note that after this assignment, and assuming that it takes no time, $|\Phi_{ma}(t^i_{sync})| \leq \epsilon_0$ and thus for any pair of clock units a, b it holds that

$$|\Phi_{ab}(t^i_{sync})| \leq 2\epsilon_0 \ .$$

In contrast, the drift adjustment is done by adapting the ratio of the frequency divider (i.e. slightly changing the value of N) in a way such that

$$\dot{vc}_a(t^i_{sync}) := \dot{vc}_a(t^i_{sync}) + \hat{\gamma}_{ma}(t^i_{sync}) \ .$$

After this drift adjustment, $|\gamma_{ab}(t^i_{sync,a})| \leq 2\gamma_0$ for every pair of clock units a, b.

7.2.2 Clock amortization vs. immediate assignment

The offset adjustment described in 7.2.1 presents an important drawback if applied as it has been defined. It causes sudden jumps of the virtual clock, which can be either backward or forward, and

such discontinuities represent a potential threat to any application in which the actions are triggered by the virtual clock. For instance, a sudden jump backwards may cause a second execution of a task recently executed.

In order to avoid these undesirable situations, the offset adjustment of OCS-CAN is not implemented as an assignment. Instead, it is progressively applied throughout the synchronization round, by shortening or lengthening a portion of the ticks in order to, respectively, accelerate or decelerate the virtual clock. In this manner, jumps do not exist and the value of the virtual clock is still monotonically increasing. This procedure is called *clock amortization*, and was described in Section 6.3.3.

As discussed in [SC90], clock amortization has no negative effect on the achievable precision, when properly implemented, and is equivalent to an immediate offset assignment. Due to this equivalence, in our analysis we assume that the offset adjustment is done by an immediate assignment, and not by clock amortization, since the former can be more easily modeled.

7.3 Analysis of OCS-CAN in fault-free conditions

In this section we further develop our mathematical modeling of OCS-CAN, and prove that in the absence of faults, any OCS-CAN system is Π-synchronized.

7.3.1 The broadcast instants vs. the synchronization instants

As already indicated in Section 6.2, the master spreads the value of its virtual clock by periodically broadcasting the TM. Any instant at which one master attempts to broadcast its TM will be called a *broadcast instant*. Since the broadcast of the TM is periodical, the broadcast instants of a master m constitute a succession of values, which will be denoted as $\{t_m^i\}$. If R is the synchronization period then the succession of synchronization instants is obtained as $\{t_m^i \mid vc_m(t_m^i) = \Delta_m + i\text{R},\ \forall\, i \in \mathbb{N}\}$, where $\Delta_m \in \mathbb{R}^+$ is a constant value that indicates the *initial phase* of master m.

The instant when the slave clock unit a receives the i-th TM broadcast by the master is called the *synchronization instant* of clock unit a. This instant, which coincides with the instant t_{sync}^i of Section 6.3.3, is denoted as $t_{sync,a}^i$ in order to keep a consistent notation.

Since any broadcast is affected by the communication channel delay, the synchronization instant will happen some time after the broadcast of the TM. For the case of CAN, the network delay is bounded by two values: the *best case response time* ($bcrt_m$) and the *worst case response time* ($wcrt_m$) [TBW95]. Therefore, any synchronization instant $t_{sync,a}^i$ satisfies that $t_m^i + bcrt_m \leq t_{sync,a}^i \leq t_m^i + wcrt_m$.

In the absence of faults, CAN guarantees that all clock units receive the TM simultaneously. This implies that in every round i, and for every pair a, b of non-faulty clock units, $t^i_{sync,a} = t^i_{sync,b}$. Note that this assumption could be hardly applicable to a software-implemented clock synchronization, because the latency caused by the message processing would be high. Nevertheless, in OCS-CAN, the message reception is handled by hardware and then the latencies are negligible.

7.3.2 Precision guaranteed in fault-free conditions

As indicated in Section 7.2.2, in our modeling we assume immediate offset and rate assignments, which cause jumps of the virtual clock. Due to this, the definition of the virtual clock as a linear function does not hold any longer. Instead, $vc_a(t)$ can be approximated by a piecewise linear function, with discontinuity points that coincide with the synchronization instants $\{t^i_{sync,a}\}$. Therefore, within the linear intervals it is still possible to apply the results of Section 7.1.3 in order to find the precision guaranteed by OCS-CAN. This is discussed in the next proposition.

Proposition 1. *Let A be a set of non-faulty clock units. In the absence of faults, the offset between any pair $a, b \in A$ of clock units is bounded as follows* $|\Phi_{ab}(t)| \leq 2\epsilon_0 + 2\gamma_0 \left(\frac{R}{1-\rho_m^{max}} + wcrt_m - bcrt_m \right)$.

Proof. We know from Section 6.3.3 that the upper bound of the absolute value of the offset between two clock units gets to a minimum right after any synchronization instant. We also know that the upper bound progressively increases as time passes by, depending on the absolute value of the consonance, until the next synchronization instant is reached and the virtual clocks are put together again. Then, to determine the precision of the system, we only have to obtain the upper bound of the absolute value of the offset right before the synchronization instants, which is

$$|\Phi_{ab}(t^i_{sync,a})| \leq 2\epsilon_0 + 2\gamma_0(t^i_{sync,a} - t^{i-1}_{sync,a}).$$

Since $t^{i-1}_{sync,a} \geq t^{i-1}_m + bcrt_m$ and $t^i_{sync,a} \leq t^i_m + wcrt_m$, we can write

$$t^i_{sync,a} - t^{i-1}_{sync,a} \leq t^i_m - t^{i-1}_m + wcrt_m - bcrt_m.$$

We know indeed that $vc_m(t^i_a) - vc_m(t^{i-1}_a) = \mathrm{R}$, and that $1 - \rho^{max}_m \leq \dot{vc}_m(t^{i-1}_m) \leq 1 + \rho^{max}_m$. Then, by Lemma 2,

$$\frac{\mathrm{R}}{1+\rho^{max}_m} \leq t^i_m - t^{i-1}_m \leq \frac{\mathrm{R}}{1-\rho^{max}_m}.$$

And therefore

$$t^i_{sync,a} - t^{i-1}_{sync,a} \leq \frac{\mathrm{R}}{1-\rho^{max}_m} + wcrt_m - bcrt_m.$$

Finally, the statement of the proposition is proved by substituting this result into the first expression.

□

Proposition 1 proves that, as long as the slaves receive the TM at periodical intervals, their virtual clocks cannot drift apart too much. Moreover, it shows that the amount of permitted offset strongly depends on the length of the synchronization period, since R $\gg wcrt_m, bcrt_m$.

7.4 Analysis of OCS-CAN with channel faults

OCS-CAN uses the low-level mechanisms provided by CAN for error detection, error signaling and error recovery; which were discussed in Section 4.3.2. Due to this, before analyzing the precision achieved by OCS-CAN in the presence of channel faults, we must discuss the failure semantics of the CAN channel.

7.4.1 Channel's failure semantics

Thanks to the error mechanisms of CAN, every frame corrupted by an error (and hence by the subsequent error frame) will be immediately retransmitted, as long as the transmitter remains non-faulty. OCS-CAN relies on this property in order to recover from errors affecting the transmission of the TM. However, it is important to remark that in OCS-CAN, whenever a TM is retransmitted, the timestamp mechanism implemented by the master clock unit overwrites the content of the data field with the timestamp of the new SOF bit. This means that the offset estimation remains as accurate as indicated in Section 7.2.1.

The arbitration mechanism implemented by CAN guarantees a bounded response time for every message broadcast, even in the presence of channel faults. Therefore, any synchronization instant $t^i_{sync,a}$ still satisfies the condition $t^i_m + bcrt_m \leq t^i_{sync,a} \leq t^i_m + wcrt_m$. Nevertheless, when channel faults are considered, the worst-case value ($wcrt_m$) increases as a consequence of the potential retransmissions.

The value of $wcrt_m$ can be estimated under different traffic and error conditions, for instance as discussed in [BBRN02]. Nevertheless, since the discussion of these methods goes beyond the scope of this document, and furthermore it is not required to understand our analysis, we will just assume

that the value of $wcrt_m$ is known for the considered fault model and traffic load, and that it is much lower than R (the synchronization period).

The potential consistency failures exhibited by CAN were discussed in Section 4.3.3. These problems were related to the appearance of either inconsistent message duplicates (IMD) or inconsistent message omissions (IMO). In the following analysis, we will assess the precision achievable by OCS-CAN, assuming the three possible consistency assumptions separately. For short, we will use the following notation: **Br-C** for Consistent broadcast; **Br-ID** for Broadcast with inconsistent duplicates; and **Br-IO** for Broadcast with inconsistent omissions. It is important to remark again that, regardless of being consistent or not, CAN still guarantees a bounded delay for any broadcast.

7.4.2 Precision with consistent broadcast

As long as the data consistency property is preserved, we know that $t^i_{sync,a} = t^i_{sync,b}$ for any pair a, b of non-faulty clock units. Thus, the only difference with respect to the error-free condition studied in Section 7.3 is the mentioned increment of $wcrt_m$ due to the potential retransmissions of erroneous frames. But these retransmissions will not have any effect on the accuracy of the offset and consonance estimations, since each retransmission of the TM contains its own timestamp. Therefore, the assumptions of Proposition 1 still hold, and the system will remain Π-synchronized, although with a greater Π. Nevertheless, since $\mathrm{R} \gg wcrt_m$ also when considering channel faults, the increment of Π is moderated.

7.4.3 Precision with inconsistent duplicates

In OCS-CAN, an additional mechanism is introduced to ensure that every slave clock unit synchronizes only to the first TM it receives in a round: after synchronizing, every slave waits for $\frac{\mathrm{R}}{2}$ time units before being available for synchronization again. In the meanwhile, every received TM is ignored. Since $\frac{\mathrm{R}}{2} > wcrt_m$, this prevents any duplicated TM from causing a second clock synchronization within the round.

Nevertheless, note that despite this mechanism it is possible for one slave to synchronize with the first TM whereas another slave synchronizes with the retransmission of that TM. Hence, under this fault assumption, the synchronization instants of two clock units may be different within the same round. However, as the next proposition proves, this circumstance does not worsen the achievable precision.

Proposition 2. *Let A be a set of non-faulty clock units that communicate through a CAN channel*

that allows inconsistent duplicates. Then, for any pair $a, b \in A$ of clock units, the offset is bounded as follows

$$|\Phi_{ab}(t)| \leq 2\epsilon_0 + 2\gamma_0 \left(\frac{R}{1-\rho_m^{max}} + wcrt_m - bcrt_m \right).$$

Proof. We start our proof by supposing that $a, b \in A$ are two slave clock units that synchronize at $t^i_{sync,a}$ and at $t^i_{sync,b} = t^i_{sync,a} + \delta$, respectively, with $\delta > 0$. Therefore $|\Phi_{ma}(t^{i+1}_{sync,a})| \leq \epsilon_0 + \gamma_0(t^{i+1}_{sync,a} - t^i_{sync,a})$ and

$$\begin{aligned} |\Phi_{mb}(t^{i+1}_{sync,b})| &\leq \epsilon_0 + \gamma_0(t^{i+1}_{sync,b} - t^i_{sync,b}) \\ &\leq \epsilon_0 + \gamma_0(t^{i+1}_{sync,b} - t^i_{sync,a}) - \gamma_0\delta. \end{aligned}$$

For both clock units, we know that the upper bound of the absolute value of the offset with respect to the master gets to its minimum at the synchronization instants, and increases monotonically after that. Therefore, the upper bound will reach its maximum possible value in both cases when $t^{i+1}_{sync,a} = t^{i+1}_{sync,b} = t^{i+1}_m + wcrt_m$. Then

$$\begin{aligned} |\Phi_{ab}(t^{i+1}_{sync,a})| &\leq 2\epsilon_0 + 2\gamma_0(t^{i+1}_m + wcrt_m - t^i_{sync,a}) - \gamma_0\delta \\ &\leq 2\epsilon_0 + 2\gamma_0(t^{i+1}_m + wcrt_m - t^i_{sync,a}) \\ &\leq 2\epsilon_0 + 2\gamma_0(t^{i+1}_m - t^i_m + wcrt_m - bcrt_m). \end{aligned}$$

As we already did in Proposition 1, we can use Lemma 2 to upper bound $t^{i+1}_m - t^i_m$, and thus the offset as follows

$$|\Phi_{ab}(t^{i+1}_{sync,a})| \leq 2\epsilon_0 + 2\gamma_0 \left(\frac{\mathrm{R}}{1-\rho_m^{max}} + wcrt_m - bcrt_m \right).$$

□

This property proves that, when only inconsistent duplicates can occur, OCS-CAN guarantees the same precision as if having data consistency.

7.4.4 Precision with inconsistent omissions

An inconsistent omission of the TM prevents at least one slave clock unit from receiving the TM. Therefore, for certain synchronization round i and clock unit a, the synchronization instant $t^i_{sync,a}$

may not be defined. This possibility makes the analysis of the precision more complicated. However, we will show that the system is still Π-synchronized, although the precision may worsen significantly.

Before starting our analysis, we need to define the concept of *consistent synchronization round*. Given a set A of clock units, we say that a synchronization round i is *consistent* if $t^i_{sync,a}$ exists for every non-faulty clock unit $a \in A$. Otherwise, we say it is *inconsistent*.

In our analysis we assume that the number of consecutive inconsistent rounds is limited. Otherwise, we should have to consider the possibility that one or more slaves might not receive the TM during an indefinite number of rounds, what would cause an unbounded increment of the absolute value of the offset with respect to the master. The parameter that bounds the number of consecutive inconsistent rounds is called the *maximum omission degree* and is denoted by $O_{max} \in \mathbb{N}$.

Thanks to the maximum omission degree, it is also possible to find the precision of OCS-CAN in the presence of inconsistent omissions. This is stated in the next proposition.

Proposition 3. *Let A be a set of clock units, and let O_{max} be the maximum omission degree of A. Then for every pair of clock units $a, b \in A$ the offset is bounded as follows*

$$|\Phi_{ab}(t)| \leq 2\epsilon_0 + 2\gamma_0 \left(\frac{(1 + O_{max})R}{1 - \rho_m^{max}} + wcrt_m - bcrt_m \right).$$

Proof. For this proof, we have to consider the worst case situation, which is whenever at least two slaves have been O_{max} consecutive rounds without receiving the TM. If a clock unit a receives the TM at $t^i_{sync,a}$ but has missed the previous O_{max} broadcasts of the TM then

$$|\Phi_{ma}(t^i_{sync,a})| \leq \epsilon_0 + \gamma_0(t^i_{sync,a} - t^{i-O_{max}-1}_{sync,a})$$

$$\leq \epsilon_0 + \gamma_0(t^i_m - t^{i-O_{max}-1}_m + wcrt_m - bcrt_m).$$

By applying Lemma 2 for bounding $t^i_m - t^{i-O_{max}-1}_m$,

$$|\Phi_{ma}(t^i_{sync,a})| \leq \epsilon_0 + \gamma_0 \left(\frac{(1 + O_{max})\mathrm{R}}{1 - \rho_m^{max}} + wcrt_m - bcrt_m \right).$$

Since $|\Phi_{ab}(t^i_{sync,a})| \leq |\Phi_{ma}(t^i_{sync,a})| + |\Phi_{mb}(t^i_{sync,a})|$, we can easily derive the boundary stated in the proposition. □

This result proves that inconsistent message omissions have stronger impact on the precision than inconsistent duplicates have. In fact, if Π_{ID} is the precision guaranteed by an OCS-CAN system

under the assumption of Br-ID and Π_{IO} is the precision guaranteed by the same system under the assumption of Br-IO then, given that R $>>$ $wcrt_m - bcrt_m$, it can be said that $\Pi_{IO} \simeq \Pi_{ID}(1 + O_{max})$.

7.5 Analysis of OCS-CAN with node faults

From a dependability perspective, implementing OCS-CAN with only one master exhibits an important drawback: if the master crashes, the TM would never be broadcast again and the precision would eventually be lost. In order to avoid this single point of failure, OCS-CAN combines two mechanisms. First, the failure semantics of the clock units is restricted to *crash failure semantics* by means of *internal duplication and comparison*. This implies that, assuming that the clock units are as simple as to be free from design errors, any internal fault of a clock unit will manifest as a crash, making the clock units *fail-silent* [Kop97]. Second, OCS-CAN defines a number of backup masters which are intended to replace the active master whenever it fails.

The purpose of this section is to analyze the effect that master crashes may have on the precision, assuming that no channel faults may exist.

In the next proposition, we show that upon master failure, and in the absence of channel faults, OCS-CAN remains Π-synchronized as long as at least one non-faulty master exists.

Proposition 4. *Let A be a set of clock units with two or more masters. Let $m \in A$ be the highest priority master and $n \in A$ be the lowest priority master. Assuming that the channel is fault-free and that there is always at least one non-faulty master, the offset between any pair of non-faulty clock units is bounded by*

$$\begin{aligned} |\Phi_{ab}(t)| \quad \leq \quad & 2\epsilon_0 + 2\gamma_0 \left(\frac{R + (\Delta_n - \Delta_m) + \epsilon_0}{1 - \rho_m^{max} - \gamma_0} \right) \\ & + 2\gamma_0 \left(wcrt_n - \frac{bcrt_m(1 - \rho_m^{max})}{1 - \rho_m^{max} - \gamma_0} \right). \end{aligned}$$

Proof. If channel faults may not exist, then $t^i_{sync,a}$ is defined for every synchronization round i and for every non-faulty clock unit a. Therefore the offset is bounded by the distance between any two consecutive synchronization instants, since $|\Phi_{ab}(t^i_{sync,a})| \leq 2\epsilon_0 + 2\gamma_0(t^i_{sync,a} - t^{i-1}_{sync,a})$. The worst case scenario would be the one in which all clock units synchronize to master m at $t^{i-1}_{sync,a}$, master m crashes afterwards and then master n takes over at $t^i_{sync,a}$. Note that this would require any other

master with higher priority than n to be faulty as well. Taking this into account, we can rewrite the boundary as $|\Phi_{ab}(t^i_{sync,a})| \leq 2\epsilon_0 + 2\gamma_0(t^i_n + wcrt_n - t^{i-1}_{sync,a})$.

In order to bound $t^i_n - t^{i-1}_{sync,a}$ we proceed in various steps. By Lemma 2 we know that if $x_l \leq vc_n(t^i_n) - vc_n(t^{i-1}_{sync,a}) \leq x_u$ and $v_l \leq \dot{vc}_n(t^{i-1}_{sync,a}) \leq v_u$ then

$$t^i_n - t^{i-1}_{sync,a} \leq \frac{x_u}{v_l}.$$

Therefore, our proof is reduced to finding the values of x_u and v_l.

To find the value of x_u we consider that $bcrt_m \leq t^{i-1}_{sync,a} - t^{i-1}_m \leq wcrt_m$, and that $1 - \rho^{max}_m \leq \dot{vc}_m(t^{i-1}_m) \leq 1 + \rho^{max}_m$. Hence, by Lemma 1

$$\begin{aligned} & vc_m(t^{i-1}_m) + bcrt_m(1 - \rho^{max}_m) \\ \leq \quad & vc_m(t^{i-1}_{sync,a}) \\ \leq \quad & vc_m(t^{i-1}_m) + wcrt_m(1 + \rho^{max}_m). \end{aligned}$$

Moreover, since master n synchronizes to master m at $t^{i-1}_{sync,a}$, then $-\epsilon_0 \leq vc_m(t^{i-1}_{sync,a}) - vc_n(t^{i-1}_{sync,a}) \leq \epsilon_0$. Therefore

$$\begin{aligned} & vc_m(t^{i-1}_m) + bcrt_m(1 - \rho^{max}_m) - \epsilon_0 \\ \leq \quad & vc_n(t^{i-1}_{sync,a}) \\ \leq \quad & vc_m(t^{i-1}_m) + wcrt_m(1 + \rho^{max}_m) + \epsilon_0. \end{aligned}$$

Furthermore, given that $vc_m(t^{i-1}_m) = \Delta_m + (i-1)\mathrm{R}$ and that $vc_n(t^i_n) = \Delta_n + i\mathrm{R}$, then

$$\begin{aligned} & vc_n(t^i_n) - vc_n(t^{i-1}_{sync,a}) \\ \leq \quad & vc_n(t^i_n) - vc_m(t^{i-1}_m) - bcrt_m(1 - \rho^{max}_m) + \epsilon_0 \\ \leq \quad & \mathrm{R} + \Delta_n - \Delta_m - bcrt_m(1 - \rho^{max}_m) + \epsilon_0 \\ \leq \quad & x_u. \end{aligned}$$

Finding the value of v_l is much easier. It has to be considered that at the synchronization instant $t^{i-1}_{sync,a}$ it holds that $-\gamma_0 \leq \dot{vc}_m(t^{i-1}_{sync,a}) - \dot{vc}_n(t^{i-1}_{sync,a}) \leq \gamma_0$. Therefore

$$\dot{vc}_n(t^{i-1}_{sync,a}) \geq 1 - \rho^m_{max} - \gamma_0 \geq v_l.$$

If we substitute the values of x_u and v_l in the expression above, we obtain the boundary initially stated. □

Note that Proposition 4 proves that a master failure would not have an important impact on the guaranteed precision. Since R $>>$ $\Delta_n - \Delta_m$, the little delay that the master replacement requires would not cause an excessive loss of precision. This can be compared with the impact of the inconsistent message omissions discussed in Section 7.4.4.

7.6 Analysis of OCS-CAN with both channel and node faults

In our previous analysis, we have shown that the fault tolerance mechanisms of OCS-CAN guarantee a certain precision both in the presence of channel faults and in the presence of node faults. Moreover, it was shown that an IMO of the TM has potentially more negative impact than other faults considered, such as IMD or master crash.

In this section, we go further in our analysis and investigate the effect that the combination of channel and node faults would have on the precision. It will be shown that an inconsistency of the TM may have greater impact when there are several masters in the system than when there is only one. Moreover, it will be shown, by means of an example, that the analysis of OCS-CAN becomes more difficult as the scenarios of inconsistency become more complex.

7.6.1 Revisiting the channel's failure semantics

Concerning channel faults, in Section 7.4.1 we distinguished three possible failure semantics, which differed in the consistency assumptions considered. Nevertheless, in the present analysis we are going to consider only one of such failure semantics: broadcast with inconsistent omissions (Br-IO). The other two hypotheses can be disregarded for the following reasons.

On the one hand, the assumption of consistent broadcast implies that the data consistency property of CAN is satisfied. As indicated in Section 7.4.2, this means that the assumptions of the analysis without channel faults are satisfied, and the bound given by Proposition 4 is still guaranteed. The only difference is a potential increment in the value of $wcrt_n$ which would be caused by the retransmissions of erroneous frames.

On the other hand, the possibility of broadcast with inconsistent duplicates is not considered because, as reported in [RVA$^+$98], any combination of faults that would cause an inconsistent duplicate would lead to an inconsistent omission if the transmitter crashes before being able to retransmit the

frame. Therefore, both possibilities, Br-ID and Br-IO, happen to be equivalent in the presence of master faults. Due to this, we only have to study the precision under the less restrictive hypothesis, which is Br-IO, and the results will be extensible to the other case.

7.6.2 Extending the concept of consistent synchronization round

Before proceeding further with our analysis, we need to define the concept of clock unit's *reference master*. Intuitively, for any synchronization round, the reference master of a clock unit is the master to which it has synchronized within that round. If a clock unit does not synchronize to any master (because no TM is received) the reference master does not change.

The reference master of clock unit a in the synchronization round i is denoted by *ref*(a, i). We assume that in the first synchronization round, all clock units synchronize to the master of highest priority, which is denoted by *hp*(M). Therefore, *ref*$(a, 0) =$ *hp*(M) for any $a \in A$. For the following rounds, *ref*$(a, i) = m$ when clock unit a receives the TM of master m at $t^i_{sync,a}$ and synchronizes to it, and *ref*$(a, i) =$ *ref*$(a, i-1)$ when no TM is received and thus $t^i_{sync,a}$ is not defined.

The value of *ref*(a, i) can be used to extend the concept of *consistent synchronization round*, which was given in Section 7.4.4, to those cases with more than one master. While using master redundancy, we say that a synchronization round i is consistent when $t^i_{sync,a}$ is defined for every non-faulty clock unit a and, moreover, *ref*$(a, i) =$ *ref*(b, i) for every pair a, b of non-faulty clock units. Otherwise the synchronization round is said to be inconsistent. The maximum omission degree (O_{max}) is still defined as the maximum number of consecutive rounds that can be inconsistent.

7.6.3 Analysis of a specific inconsistency scenario

The inconsistent synchronization rounds may represent a severe threat to the precision of OCS-CAN. This is illustrated next by means of a scenario with only one inconsistent synchronization round (i.e. $O_{max} = 1$). The analysis of this relatively simple scenario also illustrates the complexity of determining the precision when having master redundancy and inconsistent omissions.

Let us assume an OCS-CAN system such that

1. Round $i-1$ is consistent, with *ref*$(a, i-1) = m$;
2. Round i is inconsistent, with *ref*$(a, i) = m$ and *ref*$(b, i) = n$; and
3. Round $i+1$ is consistent again, with *ref*$(a, i+1) = m$.

Given that all but one of the synchronization rounds are consistent, we only need to determine the upper bound of the absolute value of the offset at the end of the inconsistent synchronization round, more specifically at $t^{i+1}_{sync,b}$. Note that in the other rounds, in which consistency is preserved, the upper bound would be lower.

From the hypothesis on round $i-1$, we know that master n synchronizes to master m at $t^{i-1}_{sync,n}$, so $|\Phi_{mn}(t^i_{sync,b})| \leq \epsilon_0 + \gamma_0(t^i_{sync,b} - t^{i-1}_{sync,n})$. Since clock unit b synchronizes to master n in round i, then $|\Phi_{nb}(t^i_{sync,b})| \leq \epsilon_o$, and therefore $|\Phi_{mb}(t^i_{sync,b})| \leq 2\epsilon_0 + \gamma_0(t^i_{sync,b} - t^{i-1}_{sync,n})$. Moreover, $|\gamma_{mn}(t^i_{sync,b})| \leq \gamma_0$ and $|\gamma_{nb}(t^i_{sync,b})| \leq \gamma_0$, so $|\gamma_{mb}(t^i_{sync,b})| \leq 2\gamma_0$. Then, we can bound the absolute value of the offset at $t^{i+1}_{sync,b}$ as

$$\begin{aligned} & |\Phi_{mb}(t^{i+1}_{sync,b})| \\ \leq\ & |\Phi_{mb}(t^i_{sync,b})| + |\gamma_{mb}(t^i_{sync,b})|(t^{i+1}_{sync,b} - t^i_{sync,b}) \\ \leq\ & 2\epsilon_0 + \gamma_0(t^i_{sync,b} - t^{i-1}_{sync,n}) + 2\gamma_0(t^{i+1}_{sync,b} - t^i_{sync,b}) \\ \leq\ & 2\epsilon_0 + \gamma_0(t^{i+1}_{sync,b} - t^{i-1}_{sync,n}) + \gamma_0(t^{i+1}_{sync,b} - t^i_{sync,b}). \end{aligned}$$

By using Lemma 1 and Lemma 2, this boundary could be expressed in terms of the clock units parameters (R, $wcrt_m$, $wcrt_n$, etc.) as it was previously done in Proposition 3 and Proposition 4.

However, even without having developed the result completely, this scenario illustrates that the effect of an IMO is more negative when using master redundancy than when there is only one master. With only one master, the offset should be bounded as stated in Proposition 3, for $O_{max} = 1$,

$$|\Phi_{mb}(t^{i+1}_{sync,b})| \leq \epsilon_0 + \gamma_0(t^{i+1}_{sync,b} - t^{i-1}_{sync,n}).$$

A comparison of both expressions reveals that, with master redundancy, the guaranteed precision has an increment of $\epsilon_0 + \gamma_0(t^{i+1}_{sync,b} - t^i_{sync,b})$, which is not negligible.

This fact must not make us erroneously deduce that using master redundancy in OCS-CAN is negative. Master redundancy is required in order to avoid having a single point of failure. Therefore this loss of precision can be seen as the price to be paid for improving the dependability of OCS-CAN. However, it is important to remark that this case perfectly illustrates how the interaction between different fault tolerance mechanisms may have unexpected effects, which need to be carefully evaluated. This is a strong motivation for performing analytical assessments as the one presented in this chapter.

Furthermore, the discussed scenario also shows that reasoning about the system behavior becomes more complex as we consider more potential faults. This growth of the complexity makes sometimes hard to ensure that, under the fault hypotheses taken into account, the worst-case scenario has been identified. This difficulty reinforces us in our previous decision of using *model checking* in order to formally verify OCS-CAN [RNPH06]. Model checking is a formal verification technique that allows

the explicit enumeration and evaluation of all possible inconsistency scenarios, and can therefore provide absolute guarantees, as it was indicated in Section 3.1.

7.7 Discussion

In this chapter we have presented a mathematical framework that allows the analytical assessment of OCS-CAN under different fault assumptions. This analysis has been addressed by approximating the virtual clocks of OCS-CAN as piecewise linear functions with discontinuity points that coincide with the synchronization instants.

The aim of our analysis was to find the equations that relate the guaranteed precision with the relevant parameters of the clock synchronization algorithm (such as the synchronization period, the number of backup masters, etc.) as well as with the considered fault assumptions. This aim has been successfully fulfilled for both channel and node faults, although we have also discovered that reasoning about the system behavior becomes more difficult when different types of faults are combined. Due to this, we advocate the application of formal verification techniques, such as model checking [CGP01], to identify and study the more complex situations. The model checking of OCS-CAN will be thoroughly discussed in Chapter 9.

In this sense, our approach for evaluating OCS-CAN is quite different from the approach followed in order to evaluate other fault-tolerant clock synchronization algorithms for CAN, such as [RGR98, HMF+00, LA03] and [APSP07]. These solutions were evaluated only by means of simulation and testing, and thus cannot provide guarantees about the behavior in different fault conditions. In OCS-CAN, although a prototype of our clock subsystem has been tested as well, we have paid more attention to the analytical assessment and to the formal verification.

The main strength of our evaluation of OCS-CAN lays on the fact that it allows a quantitative assessment of the effect that faults may have on the system precision. This allows the system designer to easily configure the system in a way such its requirements on the precision are fulfilled. Moreover, our analysis helps to understand in which conditions the desired precision may be lost and, thus, helps to identify potential risks. This could be a precious help for the design of any dependable application over CAN.

It is important to remark that in the extensive literature about fault-tolerant clock synchronization in distributed systems, it is possible to find several analyzes that study the precision of certain algorithms under different fault assumptions; for instance in [ST85, MS85, Sch87] and [FC95], among many others. Nevertheless, the results of these papers are not applicable to OCS-CAN, since they only address clock synchronization algorithms that provide fault tolerance by means of massive message

exchanges and therefore they mainly focus on proving the correctness of their *converge functions*. But their results are not extensible to any master/slave scheme, such as OCS-CAN, given that master/slave schemes use only one message per round and do not apply any converge function. The same can be said about the formal verification carried out in [PSvH99], which assumes that in every synchronization round several synchronization messages are sent and that therefore every processor counts on several timestamps per round to adjust its clock.

To the author's best knowledge, this work is the first one to explicitly address the analysis of fault-tolerant master/slave clock synchronization in the presence of inconsistent message omissions. We think that the obtained results may have an interest for the analysis of other fault-tolerant master/slave clock synchronization systems. It seems that the master/slave scheme may be the preferred solution for implementing clock synchronization in many low-cost distributed embedded systems. For instance, the clock synchronization algorithm proposed in the definition of the IEEE1588 standard [IEE02] for clock synchronization for control systems (also known as PTP-*Precise Time Protocol*) relies on master/slave, with master replication. Therefore, it is susceptible of being analyzed with the mathematical framework discussed in this document.

Chapter 8

Modeling patterns for the realistic specification of computer clocks

In Chapter 7 we carried out the analysis of the precision achieved by OCS-CAN. During said analysis, we realized that reasoning about the system's behavior becomes more complicated when considering certain combinations of channel and node faults, as it was discussed in Section 7.6. For this reason, we advocate the application of model checking in order to assess OCS-CAN for the most complex fault scenarios.

Nevertheless, the model checking of OCS-CAN is not exempt from difficulties. The most important difficulty, which was introduced in Section 3.4, concerns the limitations for specifying computer clocks by means of timed automata. Timed automata only allow, at least theoretically, specification of clocks that evolve at the rate of real time, but for the formal verification of OCS-CAN we need to specify virtual clocks that, not only evolve at a rate different from real time, but also may change their values and rates abruptly as a consequence of the synchronization actions. This circumstance forced us to adopt, and even develop, a number of modeling techniques that allowed us to circumvent these limitations and made the model checking of OCS-CAN possible. These techniques will be described in this chapter.

The problem of specifying computer clocks with timed automata is not exclusive of OCS-CAN. It has much more generality and, therefore, the modeling techniques to be discussed hereafter will actually have wider applicability. Due to this, in this chapter we will discuss these techniques from a generic perspective, i.e. as applied to a generic distributed system that uses computer clocks. Later on, in Chapter 9, we will discuss how these techniques are particularly applied for the case of OCS-CAN.

In the following sections, we will use the term *modeling pattern* to denote each modeling technique that will be described. This term stresses our focus in this chapter, which is to describe these techniques in a way that makes them reusable by other system modelers.

8.1 Contributions of this chapter

As already indicated, the problem of modeling computer clocks by means of timed automata is not exclusive of OCS-CAN. Such a problem can also be found while modeling many systems, but especially distributed systems that rely on computer clocks for coordinating their operation. Due to this generality, this problem has been addressed by several authors in different contexts, and a number of solutions are currently available [DY95, BFK+98, ADMB00, Pur00, ATM05, DL07, JBS07, WDMR08, ABG+08].

Nevertheless, despite all this previous work, it cannot be said that the specification of computer clocks with timed automata is a clearly understood issue, at least for the average system modeler. In our opinion, three circumstances have turned the modeling of computer clocks into a somehow "obscure" topic.

1. The authors of the existing solutions focus on a particular system and provide a modeling pattern suitable for that case only. They do not study the problem as a whole and then the solutions proposed are only partial.

2. The modeling of the temporal aspects of the computer clocks is very often mixed with the modeling of other aspects of the system. Due to this, some modeling patterns remain hidden for the non-expert readers.

3. Moreover, given that the authors deal with different systems and that they use different notations and strategies for modeling, it is difficult to perform a comparison of the techniques. In fact, some of the modeling techniques are complementary and, as a whole, they embrace a wide range of systems. But since they are scattered in several publications, a newcomer will have to read through many publications before knowing what can be done and, more importantly, how.

Furthermore, another difficulty that system modelers traditionally have to face is the lack of mathematical background in order to understand the papers dealing with advanced modeling aspects. For this reason, some papers that provide very valuable information are somehow "unreachable" by many system modelers. We believe that, although mathematical rigourousness is of upmost importance in the field of formal verification, it is worth to present the modeling patterns in a more comprehensible way, which should foster its application.

Table 8.1: Types of clocks and modeling patterns applied

Type of clock	Modeling techniques
Ideal clock	"Regular" timed automata
Physical clock	Perturbed timed automata
Virtual clock	Perturbed timed automata + clock pointers

This chapter tackles all of these difficulties, and presents two significant contributions to the topic of modeling computer clocks with timed automata:

1. We perform a novel description of the modeling patterns, which is both more comprehensive and (we believe) more comprehensible than previous descriptions.

2. We introduce a new modeling pattern, the so-called *clock pointers*, which makes it possible to specify distributed systems suffering from transiently inconsistent clock synchronization.

In our description we will proceed systematically. First, we will describe a simplified system model, which will constitute our case study. After that, we will introduce the modeling patterns in incremental complexity: for every type of computer clock, we will provide its corresponding modeling pattern. The techniques that constitute the basis of our modeling are summarized in Table 8.1. The concept of perturbed timed automaton was discussed in Section 3.5, whereas the concept of clock pointer will be discussed in Section 8.5 and will be further developed in Section 8.6.

For describing each modeling pattern, we will strongly rely on graphs. In our opinion, showing the graphical evolution of the timed automata clocks (and thus of the models) and comparing them to the expected temporal behavior of the systems will make the modeling patterns more comprehensible. Nevertheless, formal proofs (in the form of temporal logic properties) will be provided for demonstrating the correctness of the patterns suggested.

8.2 Description of our case study

In order to make the description of our modeling patterns easier, we will consider a simplified system model. We will essentially assume a distributed system built upon the task-based programming paradigm, since it is a widely accepted paradigm for the design of real-time systems and has enough generality. Nevertheless, it is important to remark that the modeling patterns that will be discussed

can be used as well for systems based on other programming paradigms, although perhaps with slight changes.

8.2.1 Simplified system model

In short, a *task* is a process that is activated upon the occurrence of a certain event, executes some function (or functions) and then sleeps until the next event occurrence. In this discussion we are exclusively interested in tasks that are activated by clock events and, due to this, the modeling of tasks that react to other kinds of events will not be addressed. Restricting ourselves to only clock events will help the reader to better recognize the problems to be solved when specifying computer clocks by means of timed automata, as well as to better understand the modeling patterns proposed. Once this is achieved, and according to our experience with OCS-CAN, integrating other types of events in the modeling should not be difficult.

Algorithm 1 shows a common way to program a task triggered by a clock event, also known as a *time-triggered task*. The algorithm shows a process Task1 which is activated with a fixed period T and calls the function `execute_task()` in every activation.

Note that in each activation –right before calling `execute_task()`– Task1 obtains the current value of the local clock with the function `get_time()`, calculates the next activation time for the task, and keeps it in the variable `next`. As soon as function `execute_task()` is finished, Taskl executes the sentence `sleep until next`, which makes the process enter the sleeping (or waiting) state, thus releasing the CPU, and stay idle until the next activation time. This implementation requires an underlying mechanism (typically a RTOS) to account for the time elapsed and to wake up Task1, for example with a signal, once the clock has reached the value in `next`. Other implementations are possible, for instance with programmable hardware timers, but RTOS are the most typical solution.

Algorithm 1 A possible implementation of a periodic task, with period T

```
process Task1:
loop
    next = get_time() + T
    execute_task()
    sleep until next
end loop
```

Therefore, the behavior exhibited (in theory) by Task1 corresponds to the periodical behavior de-

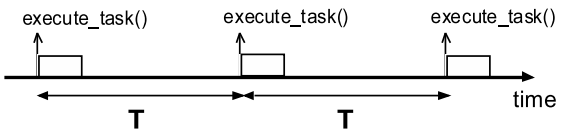


Figure 8.1: Temporal behavior of a node executing Task1 (with an ideal clock)

picted in Figure 8.1. In this figure, each small vertical arrow indicates an activation instant of Task1 and thus an instant in which the execution of function `execute_task()` begins. These activation instants are, in principle, separated by exactly T time units, so that the they occur at every time instant kT, where $k \in \mathbb{N}$ indicates the number of the round to be initiated.

Note that the specific functionality of the task is irrelevant for our modeling, and therefore the internal details of function `execute_task()` will not be considered. We just assume that the execution of this function takes some constant time shorter than T time units. The execution times are represented in Figure 8.1 with blank rectangles.

In our case study, we will consider a distributed system constituted by a number of nodes that execute Task1. We will assume that the global goal of the system is to have each node activating its own Task1 as simultaneously with the other nodes as possible. In principle, if all of the nodes had access to an ideal clock, they would execute Task1 in perfect synchrony. But this is not really the case when using computer clocks, because the individual drifts of the clocks inevitably cause lack of simultaneousness.

8.2.2 Expected temporal behavior of the system for the different types of computer clocks

It is important to remark that the function `get_time()` in Task1 does not make any assumption about the properties of the local clock, because, from the programmer's point of view, the way of reading the value of a physical clock is the same as for a virtual clock. Although this transparency is a very positive feature for modularity and code reusability, it may become a potential threat for the temporal assessment of the system. The global behavior of the system may change significantly depending on the characteristics of the computer clocks being used and, due to this, it is important to have models that *explicitly* include these characteristics.

For instance, in Figure 8.1 it was observed that the most important characteristic of Task1 was its theoretical periodicity, since `execute_task()` is called exactly every T time units. Nevertheless, this perfectly periodic behavior is a consequence of assuming and ideal clock, what means that it

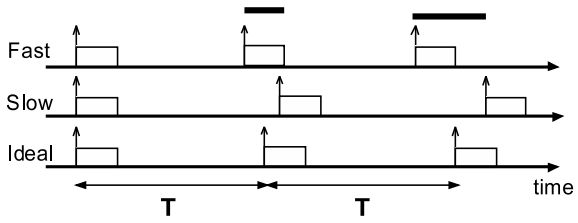


Figure 8.2: Temporal behavior of 3 nodes executing Task1 (with physical clocks)

cannot be achieved by means of real computer clocks. Next, we will describe what temporal behavior can be expected in our case study, according to the type of computer clock used by the system. This description will allow us to validate the modeling patterns that are discussed later on in this chapter.

Figure 8.2 shows one of the possible behaviors of our case study when the nodes rely on the use of physical clocks. In Figure 8.2, we represent the operation of three nodes: one of them using an ideal clock, one of them using a physical clock faster than the ideal clock, and another one using a physical clock slower than the ideal clock. Note that although the assumption of having a node that works with an ideal clock cannot be substantiated, it will be useful mainly for comparison purposes.

Again, each small vertical arrow indicates an activation instant of `execute_task()`. Note that although the three nodes coincide in the first activation of `execute_task()`, they may progressively get more and more separated. The solid black bar at the top represents the length of the interval in which `execute_task()` may be called. Thus, in other words, it is a graphical representation of the upper bound of the offset. It can the observed that the length of this bar increases in every iteration. As it was introduced in Section 7.1.2, the difference between the clock rates (the so-called *consonance*) causes a linear increment of the upper bound of the offset, which implies that nodes can get more and more desynchronized as time goes by. This effect, which is sometimes known as *clock skew*, is clearly noticeable in Figure 8.2 thanks to the solid black bar.

Figure 8.3 shows one of the possible behaviors of our case study when the nodes rely on the use of virtual clocks. In this case, we assume three virtual clocks: one that acts as the global (reference) clock, another one that is faster than the global clock and another one that is slower than the global clock. Note that, thanks to the assumed clock synchronization algorithm, the offset among the nodes is bounded by a certain value: the precision. For this reason, the solid black bar that represents the upper bound of the offset does not change its length. Then, although the tasks are not simultaneously activated, the distance between the activation instants of the same round does not increase indefinitely. This unavoidable variability on the activation instants is often known as the *jitter*.

In summary, the use of computer clocks will cause either clock skew or clock jitter in the activa-

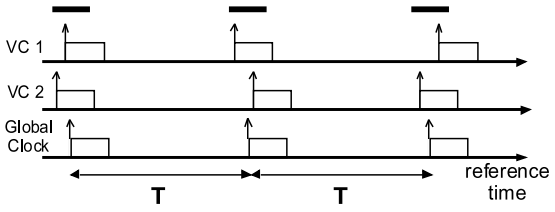


Figure 8.3: Temporal behavior of 3 nodes executing Task1 (with virtual clocks)

tion of the tasks. Therefore, system modelers require patterns for modeling these two effects when required.

8.2.3 Some remarks about modeling with timers

There are two main paradigms for the use of computer clocks in a distributed system: the *timer-driven* paradigm and the *clock-driven* paradigm [Ver94].

In a timer-driven distributed system, each node is supposed to measure time intervals locally by means of timers defined over its own computer clock. These timers are reset as required, for instance once a given event is detected. In contrast, in a clock-driven distributed system, the nodes do not use any timer internally. Instead, the computer clock of each node keeps running indefinitely and, as soon as a time mark is reached, the node performs the corresponding action and recalculates (or obtains in an equivalent way) the next time mark to wait for. Therefore, in such a system there is no need to reset any computer clock. The algorithm of Task1 shown in Section 8.2.1 is a good example of a clock-driven system.

The timer-driven paradigm can be applied either with physical clocks or with virtual clocks, whereas the clock-driven paradigm implicitly requires the adoption of a clock synchronization service, so it can only be applied to systems with virtual clocks.

Clock-driven systems represent a problem for model checking, because in a model made up of timed automata it is not possible to have clocks increasing indefinitely together with an infinite number of time marks that are to be compared with the clock values. This would make the state-space of the system blow up, and would make model checking unfeasible. Due to this, the modeler has to specify the operation of the system on the basis of rounds, with clocks that are reset in every round.

However, this round-based modeling must guarantee that, although the clocks are not allowed to increase indefinitely, the evolution of the system model still satisfies the expected temporal behavior. The round-based modeling of a clock-driven system can be performed with the modeling patterns that

will be described in this chapter, and we will show how to do it. But it is important to understand that the timers used in the model may not actually exist in the system; they are just a "modeling trick" for specifying the evolution of the nodes over time.

8.3 A modeling pattern for systems with ideal clocks

For modeling a distributed system using ideal clocks, it is enough to adopt the usual notion of UPPAAL *timer*, for instance as it is discussed in [BDL04]. These authors indicate that a timer in UPPAAL can be defined basically as a process that measures some time duration with respect to a certain UPPAAL clock, and signals the instant when that duration has elapsed.

The operation of this type of timer is simple: at a given time instant it is reset and counts up until it reaches a predetermined value. At that moment, the timer is said to have expired and generates a timeout notification. This type of timer will be called *ideal timer* hereafter, and it will constitute the basis for the description of our modeling patterns.

8.3.1 Model templates

According to [BDL04], the operation of an ideal timer can be modeled with only two primitives: set(T) and expire(), where T is a constant that indicates the duration of the timer. In principle, timers can be reset at any time instant. The timed automaton that models such a behavior is depicted in Figure 8.4(a). This timed automaton is defined with two locations, `Expired` and `Waiting`, and one clock `x`. It uses two channels for synchronization, `set` and `expire`, which correspond to the two primitives for operating the timer. Additionally, it uses a variable `T`, which may be written by other processes in order to indicate the duration of the timer.

The automaton starts in location `Expired`, in which the timer is not set. The transition to location `Waiting` may be fired at any instant via the synchronization with `set?`. Note that in this transition the value of clock `x` is reset to zero. After that, and due to the invariant and the guard defined over clock `x`, the automaton remains in location `Waiting` for exactly `T` time units, provided that the synchronization channel `set` is not activated again (since it would reset clock `x` again). As soon as clock `x` reaches the value `T`, the transition to `Expired` is fired and it is signalled through channel `expire`. Notice that in this transition a variable named `n` is increased as well. This variable will be used only for formally verifying the modeling pattern, as it will be discussed later on, but it is not really required for modeling the system.

In our case study, every node of the system will be specified by means of two timed automata: one

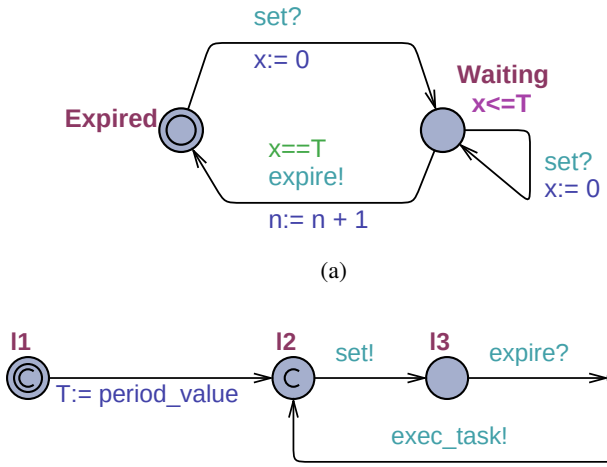


Figure 8.4: The two UPPAAL templates used for specifying the system: (a) corresponds to an ideal timer of period T whereas (b) corresponds to the application (Task1)

for modeling the timer and another one for modeling the application executed by the node. The latter is called process `App` and, in our case, it will model the execution of Task1. The timed automaton of process `App` is shown in Figure 8.4(b). This automaton has a committed initial location (`l1`) that is immediately left. In the transition to the next location (`l2`), the local variable T is updated to the value of the parameter `period_value`. Since in our case study we must force all of the nodes to activate Task1 with the same period, every process `App` will have to give the same value to its corresponding `T`.

Location `l2` is a committed location and is therefore left immediately. The transition to `l3` activates the timer, through the synchronization channel `set`. The process stays in that location until the expiration of the ideal timer, which is detected thanks to the synchronization channel `expire`. Once the expiration occurs, the process steps into the committed location `l4` and then fires the next transition to location `l2` again. The signalling through broadcast channel `exec_task` is included only for illustration purposes. At this point, it is just an abstraction of a generic function, the function `execute_task` that was discussed in Section 8.2.1. When adopting this modeling pattern, each system modeler must specify the functionality of `execute_task` according to the particularities of its own system.

Listing 8.1: Two nodes using ideal clocks (variable declaration)

```
// System variables:
const int period= 256;
const int N= 2;
chan set[N], expire[N];
int T[N];
broadcast chan exec_task;
// Observer variables:
urgent chan a;
int[0,N] n= 0;
```

8.3.2 System declaration

Once the UPPAAL templates have been defined, we proceed to explain how the rest of the system is specified. Listing 8.1 contains the variable declaration, in the UPPAAL syntax, for a system with two nodes.

The first variable defined, `period`, is a constant that indicates the period of activation of Task1. The second variable, `N`, is also a constant and indicates the number of nodes in the system; in this case, 2.

The sentence `chan set[N],expire[N]` declares two arrays of type channel, `set` and `expire`, with each array having `N` channels, i.e. one per node. The variable `T[N]` is an array of integers of `N` positions, which contains the specific period of each node. Later on, we will show the mapping procedure for assigning the channels and the period of each node. Finally, a broadcast channel `exec_task` is defined.

The last two variables declared, the urgent channel `a` and the integer variable `n`, are not required for the system itself, but for the formal verification that will be discussed in the next subsection. Their use and meaning will be clarified later on.

Listing 8.2 shows the declaration of the system, in the UPPAAL syntax again. In this system declaration, we instantiate one process `Ideal_timer` and one process `App` for each node.

In the instantiation of a process, one must indicate what template is used and which variables are mapped to the input/output parameters of every template. The template `Ideal_Timer` has three

Listing 8.2: Two nodes using ideal clocks (system declaration)

```
Timer0 = Ideal_Timer(set[0], expire[0], T[0]);
Timer1 = Ideal_Timer(set[1], expire[1], T[1]);
App0 = App(set[0], expire[0], T[0], period);
App1 = App(set[1], expire[1], T[1], period);
//List of processes to be composed into a system:
system Timer0, Timer1, App0, App1, Observer, Dummy;
```

parameters: the channels `set` and `expire`, and the variable `T`, which are used internally as it was shown in Figure 8.4(a). The template `App` has four parameters, which correspond to the channels `set`, `expire`, and to the variables `T` and `period_value`. They are used internally as shown in Figure 8.4(b).

Thanks to this mapping of parameters, `Timer0` and `App0`, whose combination constitutes one of the nodes, are synchronized through the channels `set[0]` and `expire[0]`. Conversely, `Timer1` and `App1`, which constitute the other node, are synchronized through the channels `set[1]` and `expire[1]`. Moreover, note that `App0` writes the value of `T[0]` and `App1` writes the value of `T[1]`. This means that each application configures the duration of its own timer. In our case study, we assume that each node works with the same period, so that each application will write the same value (the constant `period`) in its ideal timer.

The system declaration also includes two more processes, `Observer` and `Dummy`. These processes have no particular relationship with any system functionality, but they are used for formally verifying the system properties. This will be discussed in Section 8.3.3.

The temporal behavior that our UPPAAL model enforce is depicted in Figure 8.5. In this graph, the small vertical arrows indicate the signaling via `exec_task`. This is the moment in which each node starts its specific function, and thus these are the instants that should be synchronized.

8.3.3 Formal verification of the modeling pattern

In order to assess the properties satisfied by our model, and check whether it adheres to the expected behavior or not, we will use the verifier provided by UPPAAL. In our case, the property we wish to check is the synchronization between the ideal timers of the nodes. As depicted in Figure 8.5, they must be synchronized with no offset.

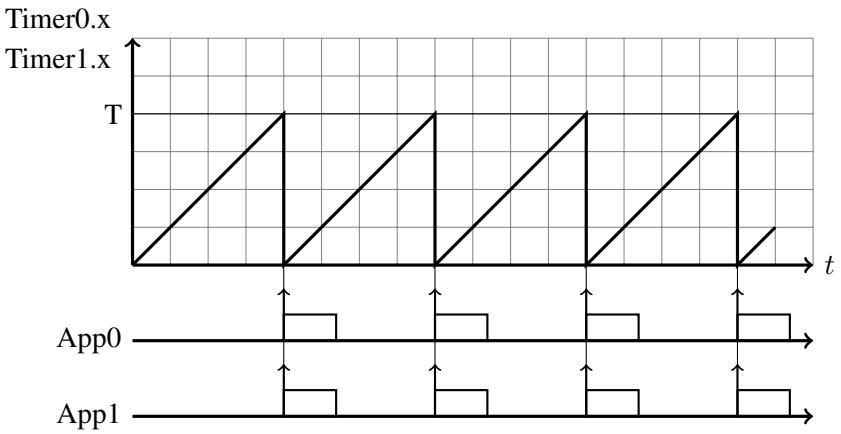


Figure 8.5: Expected temporal behavior when using ideal timers

A naive approximation to the formal verification of this property would be to evaluate the following expression:

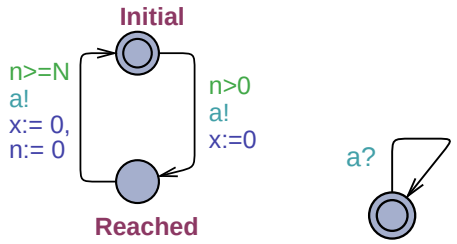
```
A[] Timer0.x == Timer1.x
```

which literally means that for all execution paths of the model, the clock `x` of `Timer0` is always equal to the clock `x` of `Timer1`. However, this kind of evaluation is not suitable for the timed automata formalism, because TA clocks cannot be reset simultaneously (two actions cannot happen at the same time) and thus there exist certain time intervals of infinitesimal length in which the value of the clocks differ.

A better way to assess the synchronization between the ideal timers is by means of a process that acts as an external observer. The timed automaton of this process, which we call `Observer`, is shown in Figure 8.6(a). Figure 8.6(b) depicts a trivial timed automaton, called `Dummy`, that process `Observer` needs in order to evolve over time.

Notice that process `Observer` starts in the location named `Initial`. The transition to the next location is enabled as soon as variable `n` is greater than 0. As discussed in Section 8.3.1, `n` is increased at the expiration instant of every ideal timer. Furthermore, the urgent channel `a` together with process `Dummy` ensure that this transition is fired *as soon as* it is enabled. Therefore, process `Observer` leaves location `Initial` and steps into location `Reached` as soon as one of the timers of the model expires. In this transition, a local clock `x` is reset.

Location `Reached` is left once the timers have all expired (condition $\texttt{n} \geq \texttt{N}$). Hence, the time spent by the `Observer` in location `Reached` is an direct measure of the amount of desynchronization among the timers. Due to this, it is possible to assess the temporal behavior of our case study by checking these two properties:



(a) (b)

Figure 8.6: The two timed automata used for verifying the precision: (a) is the Observer, (b) is a dummy automaton required for the evolution of the Observer

```
A[] not deadlock
A[] Observer.Reached imply (Observer.x == 0)
```

Satisfaction of the first property ensures that the model keeps evolving over time and is not stuck in a trivial case such that no transition is fired. Satisfaction of the second property indicates that `Timer0` and `Timer1` always expire at the same time instant, and therefore that the modeling pattern works as intended.

The strategy of having an external observer will be also adopted in order to evaluate the rest of modeling patterns that are discussed in this chapter.

8.4 A modeling pattern for systems with physical clocks

In Section 8.3 we described the modeling pattern for specifying a distributed system with nodes that have ideal clocks. Our strategy for evaluating the validity of the modeling pattern, the additional process `Observer`, was also introduced. In this section, we will propose a modeling pattern for specifying distributed system with nodes that have physical clocks. This modeling pattern uses the concept of *perturbed timed automaton* discussed in Section 3.5. The same strategy of Section 8.3 will be applied in order to formally verify the temporal properties achieved by this modeling pattern.

8.4.1 Model templates

The specification of a distributed system with physical clocks will follow the same rationale discussed for specifying a distributed system with ideal clocks. We will also use two templates for each node, one for modeling the timer and another one for modeling the application. Figure 8.7(a) shows the

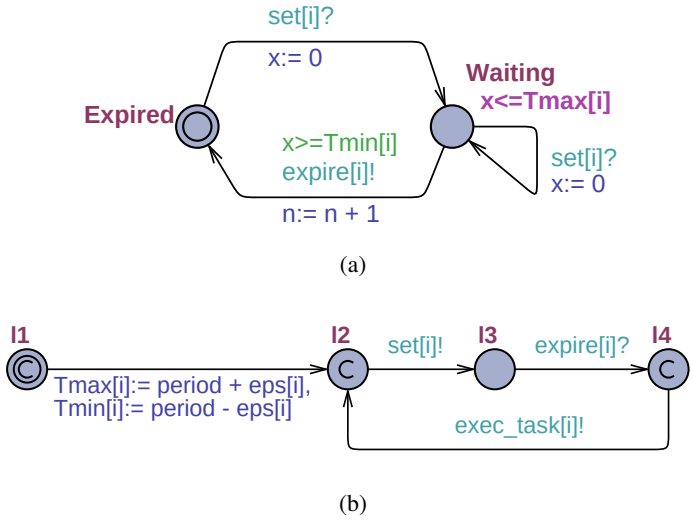


Figure 8.7: The two UPPAAL templates used for specifying the system with physical clocks: (a) is the timer and corresponds to a perturbed timed automata, whereas (b) models the application (Task1)

template for the timer, which we will call *perturbed timer*, whereas Figure 8.7(b) shows the template for the application. Notice that in both templates we introduce and index `i` to indicate which channel or variable is used. This variable `i` is passed on as a parameter of each template; what will simplify the system definition in the way discussed later on.

The main difference between the perturbed timer depicted in Figure 8.7(a) and the ideal timer shown in Figure 8.4(a) is that the perturbed timer may leave location `Waiting` at any instant within the range [`Tmin`, `Tmax`]. These two variables, which can be written by other processes, are therefore used for modeling the range of possible rates of each physical clock.

The way of obtaining the values of `Tmax` and `Tmin` was discussed in Section 3.5, when introducing the concept of perturbed timed automata. In that discussion we remarked that if the rate of a physical clock lays within the range $[1 - \rho_i, 1 + \rho_i]$ and a timer is set to measure T time units with such a physical clock, then the timer will expire at some instant between $T_{max} \simeq T(1 + \rho_i)$ and $T_{min} \simeq T(1 - \rho_i)$.

In our modeling pattern, the process `App` is intended to write the value of the variables `Tmax[i]` and `Tmin[i]`, at the beginning of the system operation. This is shown in the transition from `l1` to `l2` in Figure 8.7(b). The value of variable `eps[i]`, which is equal to $T\rho_i$, must be obtained a priori by the system modeler. The rest of process `App` does not change with respect to what was described in Section 8.3, except for the aforementioned index `i`.

Listing 8.3: Three nodes with physical clocks (variable declaration)

```
// System variables:
const int period= 300;
const int N= 3;
int eps[N] = {0, 3, 3};
chan set[N], expire[N];
int Tmax[N], Tmin[N];
broadcast chan exec_task[N];
// Observer variables:
urgent chan a;
int[0,N] n= 0;
```

8.4.2 System declaration

Listing 8.3 shows the variable declaration of the case study, assuming three physical clocks (N=3), and Listing 8.4 shows the system declaration. The idea behind our system definition is again to use one timer and one application for each node. But the way of instantiating the templates is slightly simplified with respect to the case with ideal timers.

In Listing 8.3, the declaration of three new variables should be noted. These new variables are `Tmax[N]`, `Tmin[N]` and `eps[N]`. They are defined as global variables because they are read or written by at least two processes. The value of `eps[i]` is calculated beforehand and depends on the maximum drift of the physical clock (ρ_i). In this example, we assume T $= 300$ time units, $\rho_0 = 0$ and $\rho_1 = \rho_2 = 10^{-2}$, so the variable `eps[N]` is initialized with the values $\{0, 3, 3\}$. In this example, we presuppose a null drift for the physical clock of `Timer0` only for illustration purposes. Assuming that Node 0 uses an ideal clock is not realistic, but it will help to understand the behavior of the other drifting physical clocks.

The system declaration, shown in Listing 8.4, is not very complex. Note that both templates, `Perturbed_timer` and `App`, have only one input parameter. This parameter is an integer that indicates which node the template corresponds to, and then which set of channels it must use. Again, we make process `Timer0` and process `App0` communicate via the channels `set[0]` and `expire[0]`, process `Timer1` and process `App1` communicate via the channels `set[1]` and `expire[1]`, and

Listing 8.4: Three nodes with physical clocks (system declaration)

```
Timer0 = Perturbed_Timer(0);
Timer1 = Perturbed_Timer(1);
Timer1 = Perturbed_Timer(2);
App0 = App(0);
App1 = App(1);
App1 = App(2);
//List of processes to be composed into a system:
system Timer0, Timer1, Timer2, App0, App1, App2, Observer, Dummy;
```

process `Timer2` and process `App2` communicate via the channels `set[2]` and `expire[2]`.

8.4.3 Formal verification of the modeling pattern

Figure 8.8 depicts one possible behavior of the system, according to the discussed modeling pattern. This graph is divided into two parts. In the upper part, we show the expiration instants of `Timer0`, `Timer1` and `Timer2`. Notice that `Timer0` does not drift with respect to real time and behaves like an ideal timer. As already indicated, assuming that `Timer0` uses an ideal clock is not realistic, but we included it in the example for the sake of clarity, since it makes the deviation of the perturbed timers more visible in the graph. In this example, `Timer1` is assumed to be faster than the ideal clock and `Timer2` is assumed to be slower than the ideal clock.

The lower part of Figure 8.8 shows the activation instants of the tasks, which are caused by the timer expirations. It is noticeable that, due to the lack of clock correction, the activation instants of `App1` and `App2` get more and more apart as time goes by. We represent this deviation again with a horizontal solid black bar, as it was done in Section 8.2.2.

It is important to remark that the execution of the system shown in Figure 8.8 is only one among an infinite number of possible traces; it is actually the worst case among the infinite possible cases. According to the definition of perturbed timed automata, the expirations of each timer may happen in a dense time interval, so there could be an infinite number of possible offsets among the nodes. Due to this, examination of a particular scenario has little value, and therefore we should rely on formal verification in order to examine all possible scenarios and evaluate the properties achieved by the

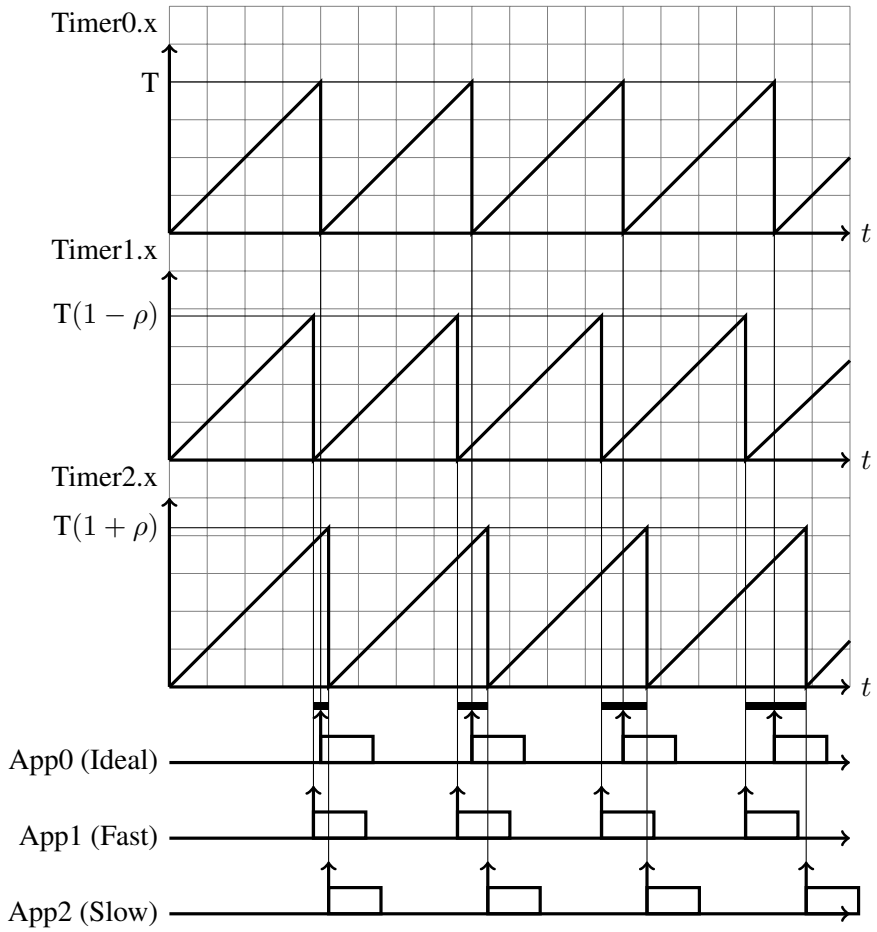


Figure 8.8: Temporal behavior enforced by the modeling pattern for physical clocks (one among infinite possible execution traces)

model.

The aim of the formal verification of this modeling pattern is to assess whether there exists an upper bound of the offset or not. As it was discussed in Section 8.2.2, the conclusion of the formal verification *must* be that the maximum offset between the activation instants is unbounded. For formally verifying this property, we use the same process `Observer` that was described in Section 8.3.3, and we model check the following property:

```
A[] Observer.Reached imply (Observer.x <= X)
```

where X is an integer. This property states that the offset between the nodes is bounded by X and therefore it should *not* be satisfied by the model for any X. Having only physical clocks implies lack of synchronization so that an upper bound for the offset cannot be defined.

The formal verification is actually performed by giving different values to X and then checking the property. The results are that for any X strictly lower than the period (in our example, $X < 300$) the property is not satisfied. Otherwise, i.e. when $X \geq 300$, the verifier outputs a verification error indicating that the property could not be checked. This result is positive, because it indicates that the modeling pattern generates so much lack of synchronization between the nodes that it causes malfunction of the whole system. In fact, it makes the round-based modeling not work properly, since the concept of round makes no sense without clock synchronization. Therefore, despite the property is not verifiable, the modeling pattern is successful in specifying the system behavior.

The properties of this modeling pattern could be investigated in more depth, but we did not proceed further because it was not required for model checking OCS-CAN. For the purpose of this thesis, it is enough to prove that it is possible to specify a system such that physical clocks drift indefinitely.

8.5 A modeling pattern for systems with virtual clocks

In this section, we will propose a modeling pattern for specifying distributed system with nodes that have virtual clocks. This modeling pattern also relies on the concept of *perturbed timed automaton* discussed in Section 3.5 but it incorporates a new technique we have called *clock pointers*. We will firstly introduce the concept of clock pointer and then we will discuss the whole modeling pattern.

8.5.1 The concept of clock pointer

The approach we have followed with the modeling patterns discussed so far is to define the temporal aspects of the system separately from the application aspects. For this reason, two different processes were defined for each node: whereas process `Timer` modeled the properties of a computer clock, process `App` modeled the application executed by each node. Additionally, the two processes belonging to the same node were communicated by means of the channels `set` and `expire`.

Then, an important characteristic of those modeling patterns is that there existed an univocal relationship between the process `Timer` and the process `App` of every node. This relationship is simply that each application sets the duration of its own timer and, once the timer expires, the application executes its task. Thus, the temporal behavior of a node is determined by its own timer, and none of the nodes can affect the behavior of the other nodes.

But such a way of modeling is only suitable for systems with either ideal clocks or physical clocks, because in such cases, the clocks are truly independent. It is unsuitable, however, for the case of having virtual clocks, because virtual clocks are synchronized and therefore they are not independent

from each other. Clock synchronization implies the existence of a reference clock, and this reference clock affects the instants in which the timer of every node is set and expires.

With respect to the model, this dependency means that we have to develop a new modeling techniques, which should allow the system modeler to choose one of the computer clocks and make it the time reference. The technique must also consider the consequences that choosing a reference clock will have on the temporal behavior of the virtual clocks. These consequences are two:

1. The reference virtual clock behaves as an ideal timer, since it cannot drift, obviously, with respect to itself

2. The other virtual clocks behave as perturbed timers, since they may deviate from the reference virtual clock. The speed at which a virtual clock deviates from the reference clock depends on the consonance (γ) between the two clocks, as explained in Section 7.1.2.

Furthermore, we need to develop a mechanism for modeling clock correction. In our model, this is achieved as follows: we force the reset of every timer to happen simultaneously with the reset of the reference virtual clock. The meaning of this action is that the offset accumulated in a certain round is corrected for the following round, regardless of its value. In this manner, whenever the reference node starts a new round, i.e. whenever it resets its timer, every other node resets its own timer simultaneously. Therefore, all nodes start every round at the same time. However, given that the virtual clocks drift from the reference virtual clock, the end of each round may be perceived by each node at a different instant.

The modeling pattern that corresponds to this behavior would be, in principle, to have one ideal timer that resets a number of perturbed timers, and to allow each perturbed timer to expire according to its own drift. Figure 8.9 depicts the temporal behavior that would be achieved with this technique, for one possible case. In this example, the virtual clock of `Timer0` is the reference time and does not drift. Note that `Timer0`, `Timer1` and `Timer2` start as soon as `Timer0` expires, but later on they expire at different instants because of their individual drifts: `Timer1` is faster; `Timer2` is slower.

As it happened with the previous modeling patterns, each task is activated by the expiration of its corresponding timer. Since the timers are modeled as perturbed timers, this modeling technique makes the activation instants of the tasks exhibit some jitter, which depends on the consonance. But the tasks do not drift apart indefinitely, as it happened when having only physical clocks. Thus, the behavior of Figure 8.9 corresponds to the expected temporal behavior of a distributed system with virtual clocks, discussed in Section 8.2.2.

Nevertheless, although this modeling pattern seems to be adequate for modeling virtual clocks, it presents a subtle flaw that prevents its application, at least in the form discussed so far. This subtle

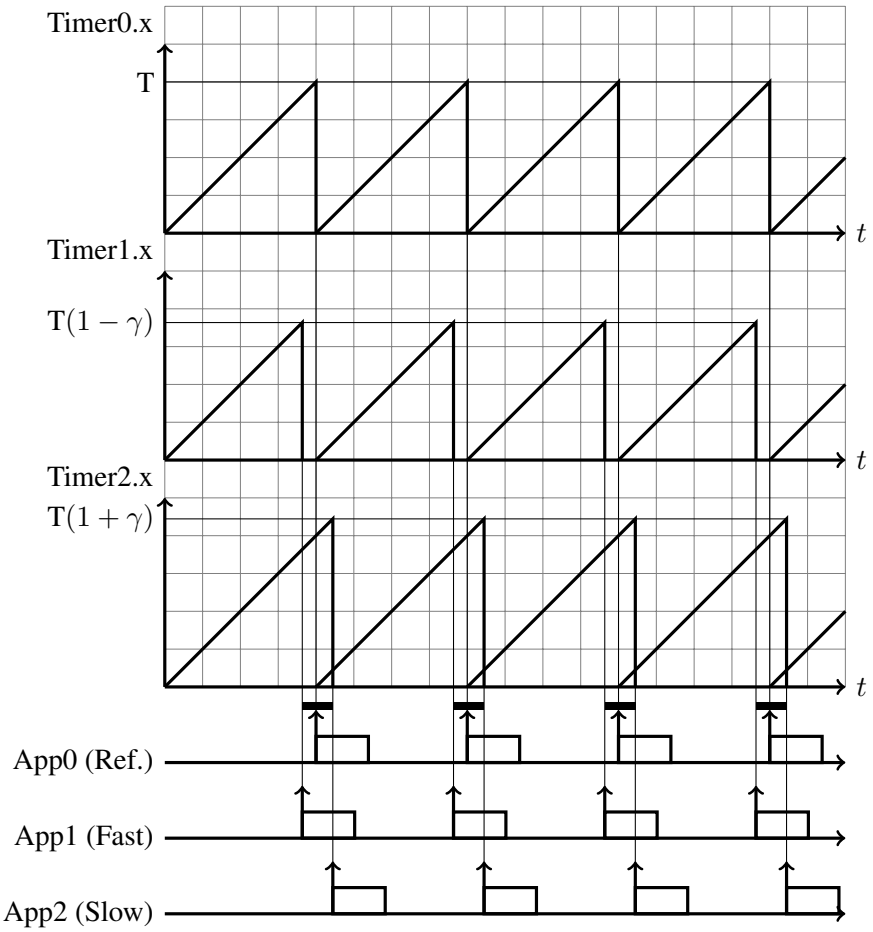


Figure 8.9: Expected behavior of three tasks using virtual clocks (not directly specifiable with timed automata)

flaw is visible for the case of `Timer2`, and is related to the fact that `Timer2` has to be reset before it has actually expired, since `Timer0` is faster and expires some time before `Timer2`. In the general case, this means that using this modeling pattern, as it has been defined, would prevent the expiration of any slower timer. Consequently, it would prevent the execution of the tasks driven by slower timers and therefore would fail to realistically model the application jitter.

Fortunately, it is possible to avoid this flaw with just a few changes in the modeling pattern. What we propose is, instead of defining a reference timer that measures T time units per round, defining a timer that measures $\frac{\mathrm{T}}{2}$ time units, but twice per round. Every segment with a nominal duration of $\frac{\mathrm{T}}{2}$ is called a Half Round. On the one hand, the first Half Round is measured and signaled by the reference timer (`Timer0` in the example), so it measures exactly $\frac{\mathrm{T}}{2}$ time units. On the other hand, the second Half Round is measured by every timer individually, so it is modeled with a perturbed

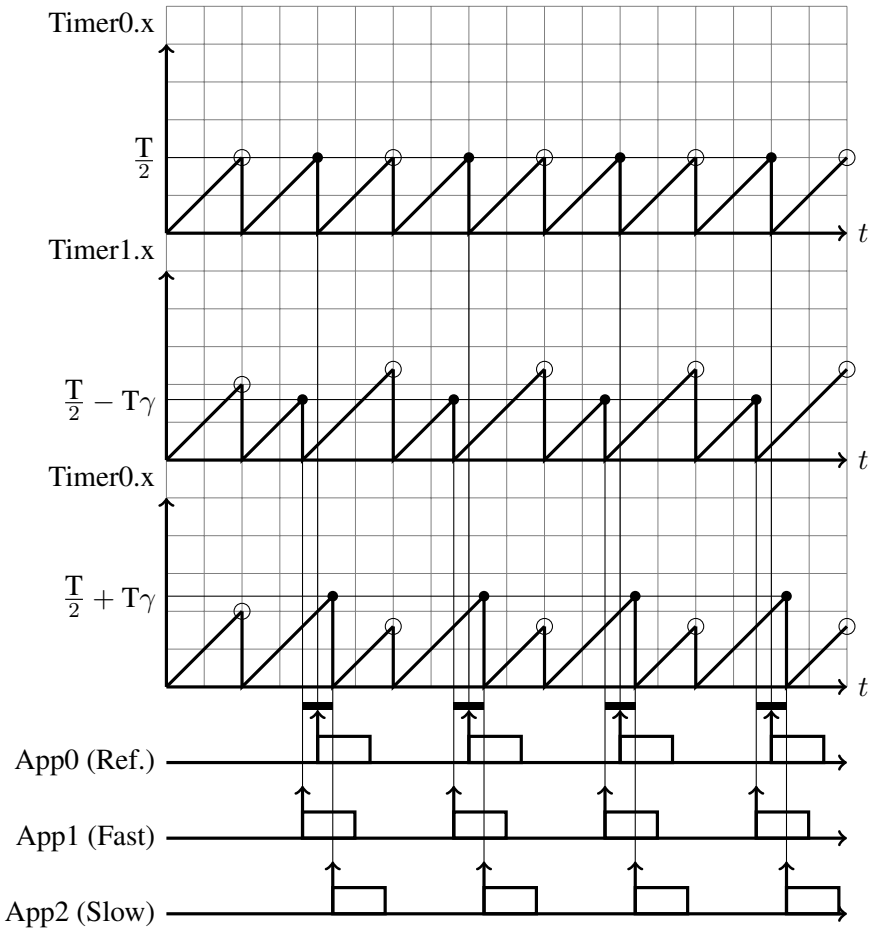


Figure 8.10: Expected behavior of three tasks using virtual clocks (directly specifiable with timed automata)

timer that expires within the interval $(\frac{T}{2} - T\gamma, \frac{T}{2} + T\gamma)$. Notice that the term $T\gamma$ accounts also for the drift accumulated during the previous Half Round.

Figure 8.10 illustrates how this modeling pattern is able to specify the expected jitter. In this example, the reference timer is defined over the clock of `Timer0`. This timer expires at the end of the first Half Segment, i.e. when the clock `x` reaches the value $\frac{T}{2}$. These instants are highlighted in the graph with empty circles.

At these instants, the other timers must be reset as well. The mechanism for signaling this event will be implemented by means of a *broadcast channel*; it will be discussed later on when describing the model templates. We will use the term *clock pointer* in order to refer to said broadcast channel, given that this channel can be seen as a pointer for accessing a timer that has been defined over a remote clock (the reference clock in this case). The empty circles appearing for `Timer1` and `Timer2`

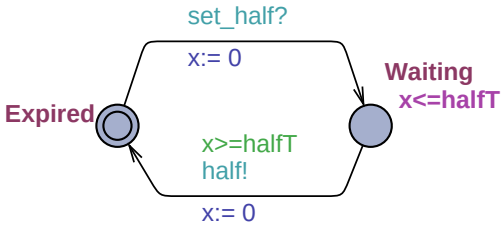


Figure 8.11: Half timer

indicate the instants in which node 1 and node 2 detect the expiration of the reference timer (thanks to the clock pointer) and hence start the second Half Round.

The duration of the second Half Round is measured by each node individually, and it is modeled as a perturbed timer that will take into account the consonance with respect to the reference clock. The black circles of Figure 8.10 indicate the expiration instants of the second Half Rounds and therefore the instants when the task must be executed by each node. It is noticeable that, as a consequence of their different consonances with respect to the reference, each Timer expires at a different instant. The example depicted in Figure 8.10 shows the worst case scenario, in which `App1` evolves as fast as possible whereas `App2` evolves as slowly as possible. Note that, even in this case, the offset between the application is always bounded.

In the next subsections, we will describe how to specify this behavior with UPPAAL and how to formally verify it.

8.5.2 Model templates

For modeling a system with virtual clocks, we will use again two processes per node, one for modeling the application (`App`) and another one for modeling the timer (`Timer`). Furthermore, we will incorporate a new process, called `Half_timer`, which will be intended to measuring and signaling the end of the first Half Round.

The timed automaton of `Half_timer` is depicted in Figure 8.11. This automaton behaves as an ideal timer: it is set through channel `set_half`, stays in location `Waiting` for exactly `HalfT` time units, and signals the expiration through the broadcast channel `half`. Therefore, channel `half` constitutes the *clock pointer* of the other nodes.

Figure 8.12(a) depicts the timed automaton of the perturbed timer used by the nodes for measuring the second Half Round. Note that it does not present any difference with respect to the perturbed timer of Figure 8.7, although the values of `Tmax[i]` and `Tmin[i]` will be different.

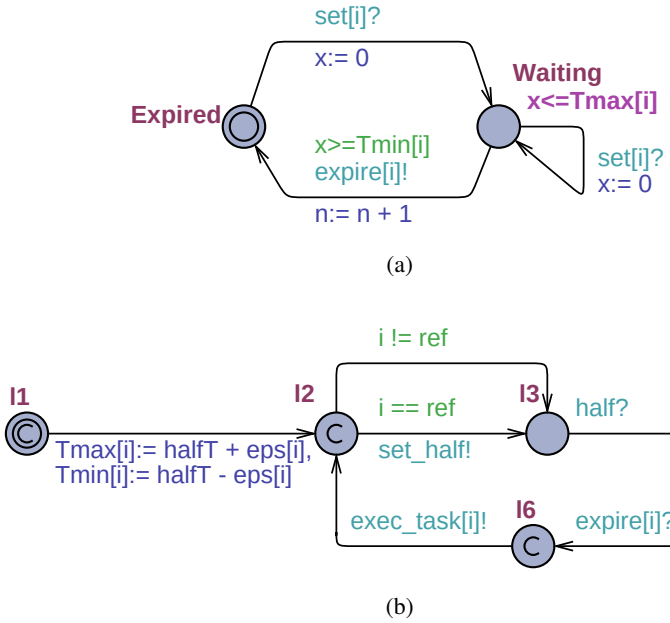


Figure 8.12: The two timed automata used for specifying the system with virtual clocks: (a) corresponds to a perturbed timed automata, and (b) corresponds to the application

The application executed by the nodes is shown in Figure 8.12(b). Notice that the first location (`l1`) is committed, so that it is left immediately. In the transition to `l2` every application sets the duration of the second Half Segment, by writing the values of `Tmax[i]` and `Tmin[i]`. The value of `halfT` equals $\frac{T}{2}$, whereas the value of variable `eps[i]`, which is calculated offline, equals $T\gamma_i$; with γ_i being the consonance of node i with respect to the reference clock. In this manner, each `App[i]` indicates not only the nominal duration of the second Half Round, but also the maximum deviation exhibited by its corresponding timer (`Timer[i]`).

Location `l2` is also committed. In the transition to `l3`, one of the nodes starts the `Half_timer`, via channel `set_half`. The node that performs this action is the one whose identifier (`i`) matches the value of a global integer variable (`ref`) that contains the identifier of the reference clock. The value of `ref` is initialized at the start of the system's operation.

All of the nodes remain in location `l3` until the expiration of the `Half_timer`, which is notified through the broadcast channel `half`. Once this happens, they step into the committed location `l4` and immediately start their own perturbed timer (via channel `set[i]`), while transiting to `l5`. As soon as `Timer[i]` expires, and signals it through `expire[i]`, the corresponding `App[i]` leaves `l5`, indicates the execution of the task (via channel `exec_task[i]`), and goes to location `l2`. After that, the whole cycle is restarted.

Listing 8.5: Three nodes using virtual clocks (variable declaration)

```
const int halfT=150; // because T= 300
const int N= 3;
const int ref= 0; // Timer 0 is the reference clock
chan set[N], expire[N];
chan set_half;
broadcast chan half;
urgent chan a;
broadcast chan exec_task[N];
int Tmax[N], Tmin[N];
int eps[N] = {0, 3, 3};
int[0,N] n= 0;
```

8.5.3 System declaration

Listing 8.5 shows the variable declaration required for specifying a system made up of three nodes using virtual clocks. In this example, the system is assumed to work with a period of T $= 300$ time units, so that `halfT` is defined as a constant equal to 150 time units. The constant `ref` is initialized to 0, meaning that Timer 0 must be taken as the clock reference of the system.

The channels `set_half` and `half` will be used for interacting with the Half Timer. The arrays of channels `set[N]` and `expire[N]` will be used for interacting with the perturbed timers that measure the second Half Round.

We assume that each virtual clock exhibits a consonance of $\gamma_i = 10^{-2}$ with respect to the reference. For this reason, both `eps[1]` and `eps[2]` are initialized with the value T$\gamma_i = 3$.

The variable declaration also includes two variables required by the observer that measures the offset: the urgent channel `a` and the integer variable `n`.

The system declaration for the same example is shown in Listing 8.6. In this declaration, we instantiate one process `HTimer` that corresponds to the template `Half_Timer` of Figure 8.11. Furthermore, we instantiate one process `Timer` and one process `App` for each node. Again, the parameter passed on to each process is an integer that indicates which set of channels is going to be used by the

Listing 8.6: Three nodes using virtual clocks (system declaration)

```
HTimer = Half_Timer;
Timer0 = Perturbed_Timer(0);
Timer1 = Perturbed_Timer(1);
Timer2 = Perturbed_Timer(2);
App0 = App(0); App1 = App(1); App2 = App(2);
// List one or more processes to be composed into a system:
system HTimer, Timer0, Timer1, Timer2,
   App0, App1, App2, Observer, Dummy ;
```

process. Therefore, it corresponds to the index `i` appearing in the timed automata of Figure 8.12.

Note that two more processes, `Observer` and `Dummy`, are instantiated. These processes are required for the formal verification of the modeling pattern, and they correspond to exactly the same timed automata discussed in Section 8.3, which were used indeed in Section 8.4.

8.5.4 Formal verification of the modeling pattern

Figure 8.13 depicts one possible behavior of the system declared above. In this graph, the upmost temporal line shows the evolution of the clock of the Half Timer (process `HTimer`). The other three temporal lines shows the evolution of the clock of the other timers.

The expiration instants of the Half Timer are highlighted with an empty circle. As explained in Section 8.5.1, this instant correspond to the end of the first Half Round and the begin of the second Half Round. Due to this, the other timers also reset their clocks at this instant. The duration of the second Half Round depends on the consonance with respect to the reference clock. In this example, `Timer0` behaves as an ideal clock and measures exactly $\frac{T}{2}$ time units, but the others behave as perturbed automata.

Each expiration of a timer that measures the second Half Round is remarked with a black circle. Note that each expiration causes the execution of the task by the corresponding process `App`. Furthermore, note that `HTimer` is reset upon the expiration of `Timer0`.

What is important of the behavior shown in Figure 8.13 is to observe that the execution of the tasks does not happen simultaneously but, nevertheless, the separation among the execution instants

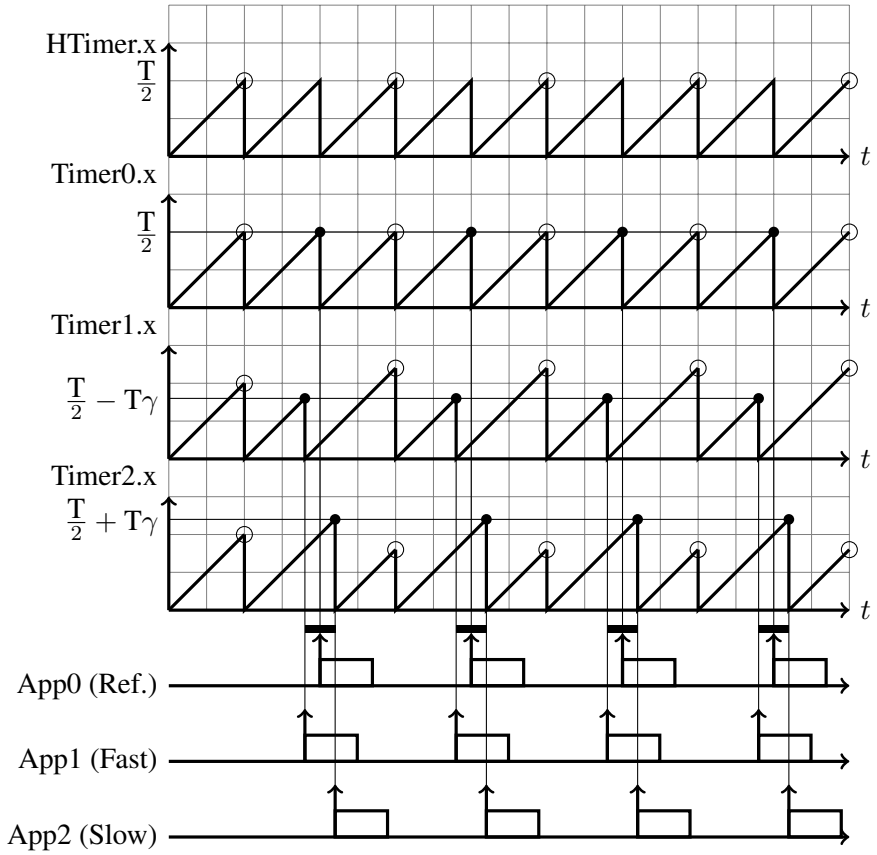


Figure 8.13: Three nodes using virtual clocks, behavior specified with a *Half Timer* (HTimer)

cannot increase indefinitely. Therefore, the application exhibits certain jitter because the nodes are synchronized with some precision. The way of formally verifying this intuitive property is by means of the process `Observer`.

In order to assess the system's behavior, for this particular example we state the following three properties:

```
A[] not deadlock
A[] Observer.Reached imply (Observer.x <= eps[1] + eps[2])
E<> Observer.Reached and (Observer.x == eps[1] + eps[2])
```

The first property checks whether the system keeps evolving over time. The second one checks whether the offset between the nodes is upper bounded, whereas the third one proves that the upper bound is reachable. The UPPAAL verifier indicates that these properties are all satisfied. Therefore, the modeling pattern is successful in specifying the intended behavior.

8.6 A modeling pattern for clock synchronization

So far in this chapter we have discussed the different types of clocks that may be found while modeling a distributed system, namely ideal clocks, physical clocks and virtual clocks, and we have proposed a number of patterns for modeling systems with such clocks. Most distributed systems will belong to one of the classes discussed in the previous sections, since they will be implemented with nodes that use either (drifting) physical clocks or (synchronized) virtual clocks; and in those cases in which assuming a perfect clock synchronization among the nodes is acceptable, using the modeling pattern with ideal clocks will be appropriate. Therefore, the modeling patterns proposed may be adopted for specifying almost the whole range of distributed applications.

Nevertheless, there is one type of distributed application that cannot be modeled with the patterns already discussed: the clock synchronization algorithms. In this section we will discuss how the modeling of these applications can addressed, and we will present a suitable modeling pattern. This modeling pattern will rely on the use of one clock pointer per node in the system.

The aim of this section is not only to describe said modeling pattern, but also to prove that thanks to the use of multiple clock pointers: 1) it is possible to specify all the types of systems discussed so far, and 2) it is possible to specify systems suffering from inconsistent clock synchronization. The latter will be further discussed in Section 9.2.6, when discussing how this modeling pattern has been adopted for the formal verification of OCS-CAN.

8.6.1 Model templates with several clock pointers

In this new modeling pattern, each node is specified by means of three processes. The name of these processes are `Half_Timer`, `Timer` and `App`; they are discussed next.

First, we define one process `Half_Timer` per node. We also define two arrays of channels, `set_half` and `half`, with the same length as the number of nodes, such that every position of the array provides a channel for interacting with one specific Half Timer. The timed automaton of process `Half_Timer` is depicted in Figure 8.14, and shows how these channels are used. It can be observed that each Half Timer behaves as an ideal timer: it is set through channel `set_half[i]`, measures the nominal duration of the first Half Round (kept in variable `halfT`), and signals the expiration through channel `half[i]`.

The process named `Timer` corresponds to a perturbed timer. The function of this process does not change with respect to what was discussed in Section 8.5; its function consists in measuring the nominal duration of the second Half Round, yet drifting with respect to the clock taken as its reference, and notifying it. For this reason, this process is specified exactly as indicated in Figure 8.12(a).

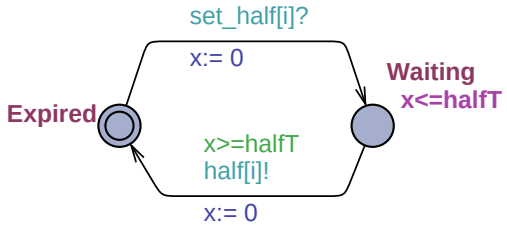


Figure 8.14: Process Half_Timer, for multiple clock pointers

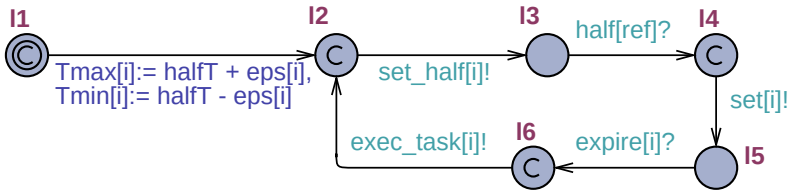


Figure 8.15: Generic application, for multiple clock pointers

Finally, process `App` is intended to executing the corresponding task as soon as a task activation instant is detected. Such activation instants are signaled by its corresponding process `Timer[i]`, through channel `expire[i]`, at the end of every round.

Note that the existence of one Half Timer per node implies that the model has as many potential clock pointers as nodes exist in the system, because each channel `half[i]` can be used by the other nodes as a clock pointer. Later on, we will show that the temporal behavior of the system is determined by the way in which these clock pointers are managed by each process `App`.

Figure 8.15 shows the timed automaton of process `App`. Although this process looks similar to the one defined in the modeling pattern for virtual clocks, there is one significant difference. In this case, every process `App` sets its own Half Timer in the transition from `l2` to `l3`, with the action `set_half[i]!`. Apart from that, there is another difference: the variable `ref` is not longer a global variable, but it is an *input parameter* that indicates which reference clock each individual process `App` is using (likewise the parameter `i`, which indicates to which specific node the process is associated). Therefore, the transition from `l3` to `l4` may be triggered by a different Half Timer in every node. Or may not, depending on what behavior the modeler wishes to specify, as discussed in the following examples.

Listing 8.7: Three nodes using physical clocks (variable declaration); multiple clock pointers

```
const int halfT=150; // because T= 300
const int N= 3;
chan set[N], expire[N], set_half[N];
broadcast chan half[N]; // potential clock pointers
urgent chan a;
broadcast chan exec_task[N];
int Tmax[N], Tmin[N], eps[N] = {0, 3, 3};
int[0,N] n= 0;
```

Example 1: modeling physical clocks

In order to model a system using *physical clocks*, the system modeler must make every process `App` work with its own Half Timer as the time reference. In other words, each node must work with its own clock pointer, which must be independent of the others' clock pointers. We will illustrate this concept with an example.

Listing 8.7 shows the variable declaration of a system having 3 nodes. With respect to the variable declaration discussed in Section 8.5, there are only two differences, both already mentioned: `set_half` and `half` are defined as two arrays of channels of length `N`, and the global variable `ref` is not declared.

In this example we assume that Node 0 is provided with an ideal clock and, for this reason, the value of `eps[0]` is 0. This assumption helps us to introduce the modeling pattern, since it simplifies the graphs, but it is not required a priori. The other two nodes are assumed to have physical clocks that drift with respect to real time, with $\rho = 10^{-2}$ for each physical clock.

The system declaration is shown in Listing 8.8. Note that, as already indicated, we instantiate three processes for each node. Again, the first parameter of each process corresponds to the node's identifier. Therefore, it sets the relationship among the three processes that are part of the same node. However, the most important part of this system declaration is the second parameter of process `App`. This parameter indicates which `Half_Timer` is going to be followed by the application. Then, this number sets the relationship between the process and its reference clock.

Since we wish to model a system with physical clocks, and physical clocks are independent from

Listing 8.8: Three nodes using physical clocks (system declaration); multiple clock pointers

```
HTimer0 = Half_Timer(0); HTimer1 = Half_Timer(1);
HTimer2 = Half_Timer(2);
Timer0 = Perturbed_Timer(0); Timer1 = Perturbed_Timer(1);
Timer2 = Perturbed_Timer(2);
App0 = App(0,0); App1 = App(1,1); App2 = App(2,2);
// List one or more processes to be composed into a system:
system HTimer0, HTimer1, HTimer2, Timer0, Timer1, Timer2,
    App0, App1, App2, Observer, Dummy ;
```

each other by definition, then each process `App` is forced to follow its own Half Timer. That is the reason why, for every `App`, the second input parameter (the reference) coincides with the first input parameter (the node's identifier).

The temporal behavior resulting from this modeling pattern is depicted in Figure 8.16. In this graph we represent the evolution of the clock of each process `HTimer` together with the evolution of the clock of each process `Timer`, as well as the activation of its task by each process `App`.

The empty circles highlight the expiration instants of the Half Timers, which are signaled through the corresponding broadcast channel `Half[i]`. Notice that each process `Timer` uses the expiration instant of its peer Half Timer for resetting its own clock (this could be compared to the graph in Figure 8.13, where all nodes use the same reference Half Timer). After that, each process `Timer` measures the second Half Round, but exhibiting a certain drift with respect to real time; except for `Timer0`, which is assumed to work with an ideal clock. In this graph, we assume that Node 1 evolves as fast as possible and Node 2 evolves as slowly as possible.

As a consequence of following different clock pointers (i.e. different Half Timers), the applications inevitably drift apart, and the execution instants of the tasks get more and more separated. The upper bound of the offset is represented graphically by means of a solid black bar. The length of this bar increases in every round.

At first sight, the behavior depicted in Figure 8.16 is equivalent to the behavior shown in Figure 8.8. Moreover, if we try to model check whether the offset among the physical clocks is bounded or not, we will obtain the same verification error discussed in Section 8.4.3, indicating that the property could not be verified. This means that the pattern correctly models the expected behavior of a system using

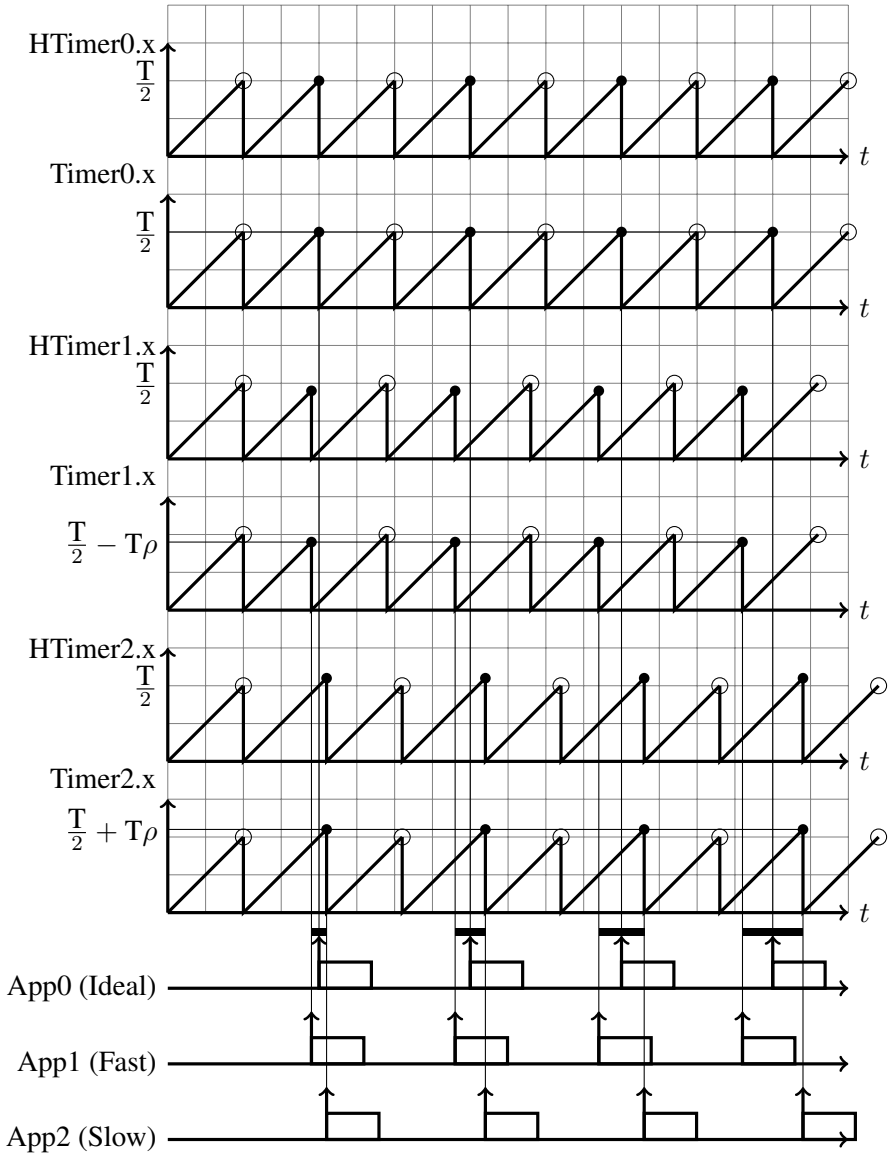


Figure 8.16: Three nodes using physical clocks, behavior specified with one *Half Timer* per node

drifting clocks.

Example 2: modeling virtual clocks

In order to model a system using *virtual clocks*, the system modeler must force all of the nodes to use the same clock pointer. The system declaration of Listing 8.9 shows how this can be done. Note that the only difference with respect to the system declaration of Listing 8.8 is the way of instantiating the

Listing 8.9: Three nodes using physical clocks (system declaration); multiple clock pointers

```
HTimer0 = Half_Timer(0); HTimer1 = Half_Timer(1);
HTimer2 = Half_Timer(2);
Timer0 = Perturbed_Timer(0); Timer1 = Perturbed_Timer(1);
Timer2 = Perturbed_Timer(2);
App0 = App(0,0); App1 = App(1,0); App2 = App(2,0);
// List one or more processes to be composed into a system:
system HTimer0, HTimer1, HTimer2, Timer0, Timer1, Timer2,
   App0, App1, App2, Observer, Dummy ;
```

processes `App`. In this example, the second input parameter gets the value 0 for all cases, meaning that Node 0 must be taken as the reference clock.

It is important to remark that this example uses the same variable declaration of Example 1 (already shown in Listing 8.7). Although in this case the value of `eps[i]` depends on the consonance with respect to the reference virtual clock and not on the drift with respect to real time. So, in fact, we are assuming $\gamma = 10^{-2}$ for `Timer1` and `Timer2`.

The temporal behavior achieved with this modeling pattern is shown in Figure 8.17. As expected, the behavior is equivalent to the one achieved in Section 8.5, which was depicted in Figure 8.13. The solid black bar illustrates the fact that the tasks are executed with a certain jitter. Furthermore, the properties stated in Section 8.5 are also satisfied.

8.6.2 How to extend the modeling pattern for including clock synchronization

It is important to remark that the modeling patterns that have been already discussed for specifying systems with virtual clocks do not really model clock synchronization. They only model the *jitter* of the virtual clocks with respect to the reference clock. But, although the jitter is a consequence of having clock synchronization, it is not the clock synchronization mechanism itself. All the internal details regarding clock synchronization are actually hidden, so we can say that the modeling patterns for virtual clocks discussed so far are just *abstractions* of the service supplied by the clock synchronization algorithm.

However, modeling the internal details of the clock synchronization algorithm is required some-

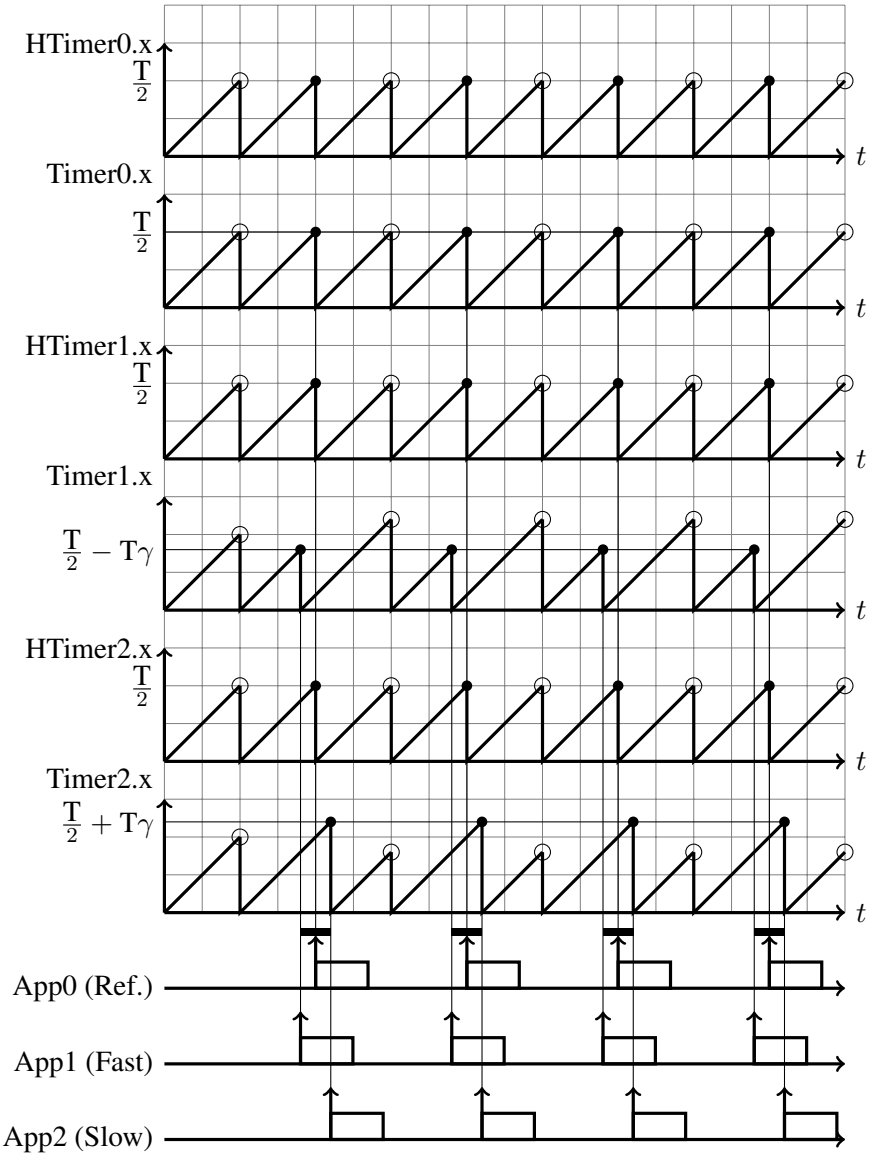


Figure 8.17: Three nodes using virtual clocks, behavior specified with one *Half Timer* per node

times. For instance, we need to model the internal details of the clock synchronization algorithm if we wish to verify whether the internal mechanisms work as intended and guarantee the desired precision. In those cases, the modeler cannot assume a certain precision a priori (as it is done with the modeling pattern for virtual clocks) but should, instead, use a modeling pattern that allows changes of the offset according to the evolution of the system. This is exactly the case of OCS-CAN.

In the formal verification of OCS-CAN, our aim will be to model check how certain faults of the nodes and of the channel may affect the precision achievable. More specifically, we will be

mainly interested in assessing the effect that the inconsistent synchronization rounds may have on the offset of the virtual clocks, because, as it was explained in Chapter 7, these failures are particularly threatening for the precision. Therefore, given that we wish to find out an upper bound for the offset (the precision), the starting point cannot be a model in which the offset is bounded by default.

The modeling pattern we have developed for OCS-CAN is based upon the idea of having multiple clock pointers, since this technique allows the co-existence of several reference clocks at the same time. However, we will extend the modeling pattern discussed before, and will allow the modeler to dynamically change the offset and the consonance between the virtual clocks when required.

We will illustrate this concept next, by modeling a distributed system consisting of four nodes that implement a master/slave clock synchronization algorithm.

Description of the case study

In our case study we consider a distributed system made up of four nodes. Every node may become a master, but we assume that they are hierarchically organized and in fault-free conditions they all follow the highest-priority node. Node 0 is the one with the highest priority, whereas Node 3 is the one with the lowest priority. A node may only follow a master that has higher priority than itself, never a node with lower priority.

The clock synchronization algorithm implemented by the nodes is a periodic process, executed in rounds of duration T. At the end of every round, each node selects one node as its reference clock and adjusts its own clock in order to follow said clock. These instants therefore correspond to the *synchronization instants* that were defined in Section 7.2.1. These instants are denoted as t^i_{sync}, where i indicates the synchronization round.

As it was also explained in Section 7.2.1, every node performs two actions at each synchronization instant: an offset adjustment and a drift adjustment. As a consequence of these actions, two properties are verified: (1) $|vc_m(t^i_{sync}) - vc_a(t^i_{sync})| \leq \epsilon_0$, and (2) $|\dot{vc}_m(t^i_{sync}) - \dot{vc}_a(t^i_{sync})| \leq \gamma_0$; where m is the chosen master and a is the node synchronizing to the master.

Unless otherwise stated, we will hereafter assume that ϵ_0 is negligible. Therefore, after a synchronization action, and for every node a synchronizing, the absolute value of the offset with respect to the reference master that will accumulate in the nominal duration of a round (T) is at most $|\Phi_{ma}| = \mathrm{T}\gamma_0$. In the modeling templates discussed next, the value $\mathrm{T}\gamma_0$ will be denoted as e0.

Although ϵ_0 could be included in the model, actually by making $\mathrm{e0} = \epsilon_0 + \mathrm{T}\gamma_0$, it does not have too much relevance for the assessment of the precision. Especially for hardware-implemented clock synchronization systems, such as OCS-CAN, because the high accuracy of the timestamp mechanism

and the low latency of the offset adjustment reduce the value of ϵ_0 significantly.

In fault-free conditions, the nodes should all choose the same master (the one with the highest priority, Node 0) in every round and then always follow the same clock reference. The behavior enforced in that way should be like the behavior obtained when modeling virtual clocks, for instance in Figure 8.17. However, in our case study we will introduce the possibility of having inconsistent synchronization rounds. We will use the definition of consistency that was discussed in Section 7.6.2: we say that a synchronization round i is *consistent* if all nodes choose the same master, otherwise the synchronization round is said to be *inconsistent*. We will adopt as well the concept of *maximum omission degree* (O_{max}), which in Section 7.6.2 was defined as the maximum number of consecutive rounds that can be inconsistent.

In order to model potential inconsistencies, at the end of every round we will make each node indeterministically select one master among the nodes that do not have lower priority; what includes the node itself as well as the nodes with higher priority. The reasons for choosing one specific master have no interest for the purpose of this chapter and thus will be disregarded. Even though they are important in a realistic case and thus will be thoroughly discussed when addressing the model checking of OCS-CAN in Chapter 9.

The model will include an external process that will count the number of consecutive inconsistent rounds. Whenever this count reaches the value O_{max}, the external process will disable the possibility that the following round be inconsistent again.

Model templates

In our modeling pattern for clock synchronization, each node is specified by means of three processes, namely `Half_Timer`, `Timer` and `App`.

The process `Half_Timer` and `Timer` are used for measuring the periodic synchronization rounds, of period T, in a similar way as it was discussed in Section 8.6.1. I.e., the former measures the first half of the round and the latter measures the second half of the round.

Process `Half_Timer` is modeled with the same timed automaton of Figure 8.14 and behaves like an ideal timer. In contrast, the timed automaton of process `Timer` changes slightly. This process, whose timed automaton is depicted in Figure 8.18, implements a perturbed timer that measures a nominal duration of `halfT` time units, but with certain deviation represented by the global variable `eps[i]`.

The value of the variable `eps[i]` may change in every synchronization round. This feature of the model constitutes an important difference with respect to the perturbed timers used in the previous

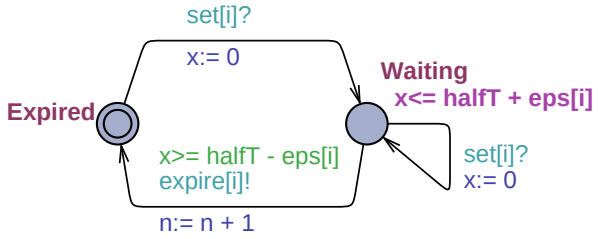


Figure 8.18: A Perturbed Timer for modeling clock synchronization

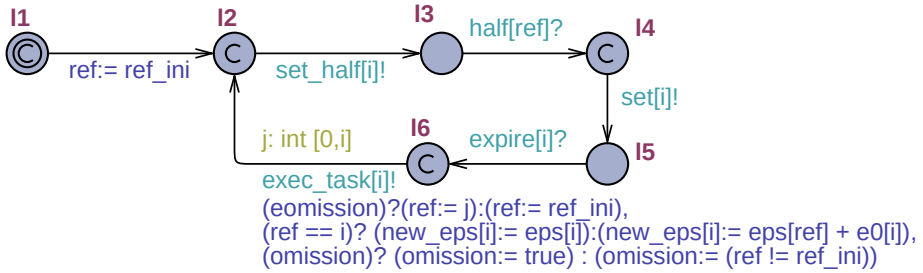


Figure 8.19: An application that models master/slave clock synchronization

modeling patterns: in the former we assumed that the variation interval of the perturbed timers had a fixed length, whereas now we allow these intervals to change dynamically. The amount of variation is determined by the synchronization application, which is modeled by process `App`.

It is important to remark that the clock of the highest priority master (in this case, Node 0) constitutes the reference clock of the whole system, and therefore every perturbed timer `Timer` is defined as if this clock provided the value of real time. Then, even though one node may synchronize to a master that is not Node 0, the offset `eps[i]` of the corresponding `Timer` will be still measured with respect to Node 0.

Figure 8.19 shows the timed automaton of process `App`. The first important feature of this timed automaton is that `ref` is not an input parameter, but a local variable. It has to be a local variable because it may change during the system's operation. The initial value of `ref` is passed on as an input parameter named `ref_ini`. Notice that `ref` is initialized to `ref_ini` in the first transition, from location `l1` to `l2`. In our case study, the value of `ref_ini` will be 0, since it is the identifier of the highest-priority node.

A second important feature of this timed automaton is that the value of `ref` may change in the transition that occurs at the end of each round; in the transition from `l6` to `l2`. This transition takes advantage of the *selection* feature provided by UPPAAL, which makes a nondeterministic choice of a

value within the range [0, i] and assigns it to a temporal variable named `j`, which is visible only in the *update* associated to the transition. Note that such a range of possible values corresponds to the identifiers with equal or higher priority than the node itself.

In order to simplify the comprehension of the actions performed by process `App` in the transition from `l6` to `l2`, in Algorithm 2 we show the algorithm specifically implemented. This algorithm starts with the nondeterministic assignment of a value to `j`. After that, and provided that omissions are allowed for the next round, a number of variables are updated, as it will be described next.

Algorithm 2 Algorithm for choosing and assigning the master

```
j ← x ∈ {0, i}
if eomission then
    ref ← j
else
    ref ← ref_ini
end if
if ref = i then
    new_eps[i] ← eps[i]
else
    new_eps[i] ← eps[ref] + e0[i]
end if
if omission then
    omission ← TRUE
else
    omission ← ( ref != ref_ini )
end if
```

The possibility of having an inconsistent synchronization round is enabled by a global boolean variable named `eomission`. If `eomission` is TRUE then `ref` takes the value of `j`; if not, then `ref` takes the value of `ref_ini`. This implies that whenever `eomission` is FALSE, all of the nodes will choose `ref_ini` as the reference master and therefore the synchronization round will become consistent.

After the master has been chosen, the offset with respect to the reference time (kept in variable

new_eps[i]) has to be updated accordingly. If the node has synchronized to itself, condition ref=i, then the new offset does not change. If the node has synchronized to another master, then it inherits the offset with respect to ref_ini of said master, which is kept in the variable eps[ref], and adds its own offset to the selected master. This additional offset is equal to e0, because, after the clock adjustment takes place, the consonance with respect to the chosen master is γ_0. The value of e0 might be different for each node, and is kept in the global variable e0[i].

Finally, the node updates a global boolean variable called omission for registering whether an inconsistent synchronization has happened or not. In case that the node has actually synchronized to a master that is not ref_ini, then omission takes the value TRUE. Once this variable is set to TRUE it cannot be set to FALSE again by any other node. As we will explain next, this has to be done by an external process, the so-called Observer, at the end of each round.

The timed automaton of process Observer is depicted in Figure 8.20. This process performs two functions. The first one was already described when introducing the previous modeling techniques, and is related to the assessment of the precision. Local variable n indicates the number of nodes that have reached the end of the round. Thus, the invariants defined over this variable guarantee that process Observer enters location Reached as soon as the first node ends its round and leaves said location as soon as all of the nodes have reached the end. In this way, the time that process Observer may remain in location Reached gives the maximum offset between the nodes and thus the guaranteed precision. An implicit assumption of this mechanism is that the offset between the nodes never exceeds the length of a Half Round, i.e. $\frac{\mathrm{T}}{2}$ time units.

The second function of process Observer concerns the management of the state of the current and the next synchronization rounds. Note that whenever all of the nodes have reached the end of the round, it is guaranteed that each node has chosen its own reference clock for the next round. Therefore, the transition from Reached to Initial is the right instant for (1) determining whether the round has been consistent or not, and (2) updating the offset for the next round, eps[i], of every node. This offset depends on which specific reference the node has chosen.

In order to determine the state of the just-finished round, process Observer checks the value of the global boolean variable omission. Remember that said variable takes the value TRUE if at least one of the nodes selected a master different from ref_ini. Then, whenever this variable equals TRUE, the number of consecutive inconsistent rounds (kept in the global integer variable o_count) is increased by one unit. Otherwise, o_count is set to 0.

After updating o_count, its value is compared to O_{max}. In case they are equal, the variable eomission is set to FALSE, thus disabling the possibility of an inconsistency during the next synchronization round. Then, the variable omission is set to FALSE, so that the nodes will again be able to update the state of the next synchronization round.

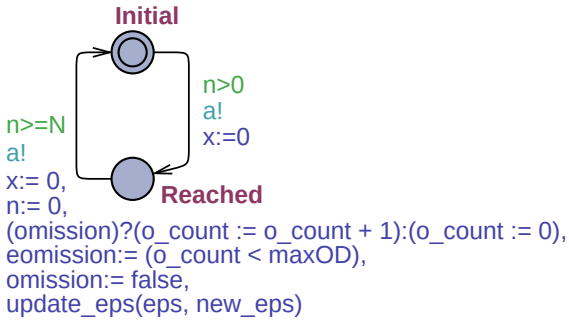


Figure 8.20: New process Observer, which also updates the offset of each node after every round

For updating the offset of the nodes, process `Observer` calls a function specifically designed for this purpose. The name of this function is `update_eps()` and has two input parameters: a pointer to the array `eps` and a pointer to the array `new_eps`. The declaration of this function is discussed next, together with the rest of the system declaration.

System declaration

Listing 8.10 shows the variable declaration for our case study. This declaration does not differ too much with respect to other declarations already discussed. The main differences are: the declaration of constant `maxOD`, which stands for O_{max}; the declaration of the arrays `e0[N]` and `new_eps[N]`, whose utility was discussed together with the model templates; the declaration of the variables `e_omission`, `o_count`, `omission` for managing inconsistencies; and the declaration of the function `update_eps()`.

Notice that function `update_eps()` is declared with a C-like notation. It takes two array pointers as input parameters, namely `a` and `b`. Then for each position of `b`, it makes `a[i] = b[i]` and sets the value of `b[i]` to 0. In other words, this function overwrites the array `a` with the values of `b` and then resets `b`.

The system declaration of the case study is shown in Listing 8.11. In this declaration, we instantiate three processes for each one of the four nodes. The first input parameter is the node's identifier, parameter `i` in the model templates. The second input parameter that appears in the instantiation of process `App` is the value of `ref_ini`, so that it takes the value 0 (the identifier of the highest priority node) for each process.

Listing 8.10: Four nodes using virtual clocks (variable declaration)

```
const int halfT=150; // halfT= period/2
const int N= 4;
const int maxOD= 3;
chan set[N], expire[N], set_half[N];
broadcast chan half[N];
urgent chan a;
broadcast chan exec_task[N];
const int e0[N] = {0,3,3,3};
int eps[N] = {0,3,3,3}; //initially synchronized to 0
int new_eps[N] = {0,0,0,0};
int[0,N] n= 0;
bool e_omission= (maxOD > 0);
int[0, maxOD] o_count= 0;
bool omission= false;

void update_eps(int& a[N], int& b[N]) {
 for (i : int[0,N-1]) {
   a[i] = b[i];
   b[i] = 0;
   }
}
```

Listing 8.11: Four nodes using virtual clocks (system declaration)

```
HTimer0 = Half_Timer(0); HTimer1 = Half_Timer(1);
HTimer2 = Half_Timer(2); HTimer3 = Half_Timer(3);
Timer0 = Perturbed_Timer(0); Timer1 = Perturbed_Timer(1);
Timer2 = Perturbed_Timer(2); Timer3 = Perturbed_Timer(3);
App0 = App(0,0); App1 = App(1,0);
App2 = App(2,0); App3 = App(3,0);
// List one or more processes to be composed into a system.
system HTimer0, HTimer1, HTimer2, HTimer3,
    Timer0, Timer1, Timer2, Timer3,
    App0, App1, App2, App3, Observer, Dummy ;
```

Study of the temporal behavior specified

Once the model templates and the system declaration have been discussed, we can study which kind of temporal behaviors this modeling pattern can specify. For that purpose, we will consider some representative examples.

First of all, we will study the trivial case in which we define both variables `e0` and `maxOD` as equal to 0. This configuration implies that the virtual clocks do not drift from each other and that inconsistent rounds are not possible. Therefore, it corresponds to a system using ideal clocks and should exhibit the same temporal behavior.

It is possible to determine that the modeling pattern enforces the desired temporal behavior. The UPPAAL verifier indicates that the following two properties are satisfied:

```
A[] not deadlock
A[] Observer.Reached imply (Observer.x == 0)
```

As indicated in Section 8.3, the fulfillment of these properties proves that the system behaves as if using only ideal clocks.

The second case we are going to study is the case in which the virtual clocks may drift from each other (we define `e0` greater than 0) but inconsistencies are not allowed (we make `maxOD`= 0). This case corresponds to a system using virtual clocks as discussed in Section 8.5, and also in the

Example 2 of Section 8.6.1.

Again, we rely on the UPPAAL verifier for determinining what temporal behavior is enforced with this modeling pattern. We obtain that the following three properties are satisfied:

```
A[] not deadlock
A[] Observer.Reached imply (Observer.x <= 2*e0[1])
E<> Observer.Reached and (Observer.x == 2*e0[1])
```

This means that the jitter of the application, characteristic of a system with consistently synchronized virtual clocks, is adequately modeled. Note that in this example we assume, without losing generality, that all `e0[i]` are equal; except for `e0[0]`, which takes the value 0 because the reference clock cannot drift with respect to itself.

Nevertheless, the potential of this new modeling pattern can only be observed if we check more complex scenarios. For instance, consider the example appearing in Figure 8.21. In this graph we show the temporal behavior of a system made up of four nodes, in which Node 1 and Node 2 take Node 0 as the clock reference and follow it, whereas Node 3 runs free and drifts away from Node 0 (and hence from the other nodes as well).

As a consequence of the lack of synchronization of Node 3, the upper bound of the offset increases. This is highlighted in the graph by means of an additional horizontal black bar, which indicates the deviation between process `App3` and the other processes `App`. The length of this additional black bar increases as time goes by, because Node 3 is not able to synchronize to Node 0. Although it is not shown in the graph, as soon as Node 3 synchronizes to Node 0, it corrects its deviation and gets together with the other nodes.

Note that the scenario of Figure 8.21 requires the inconsistent synchronization of Node 3 during three rounds. Therefore it can be specified with the presented modeling pattern by making `maxOD`= 3. However, it is important to remark that the depicted scenario is only one possible case of inconsistency when `maxOD`= 3. More complex scenarios can exist, for instance if Node 2 also runs free, to give just an example.

A very positive aspect of our modeling pattern is that it guarantees that all possible scenarios are generated and model checked. Moreover, for finding out the upper bound of the offset, we do not need to know the analytical expression that relates the values of the virtual clocks. Instead, we make the UPPAAL verifier model check the following properties, but giving different values to X.

```
A[] not deadlock
A[] Observer.Reached imply (Observer.x <= X)
E<> Observer.Reached and (Observer.x == X)
```

The results obtained are shown in Table 8.2. These results prove that there is a strong relationship

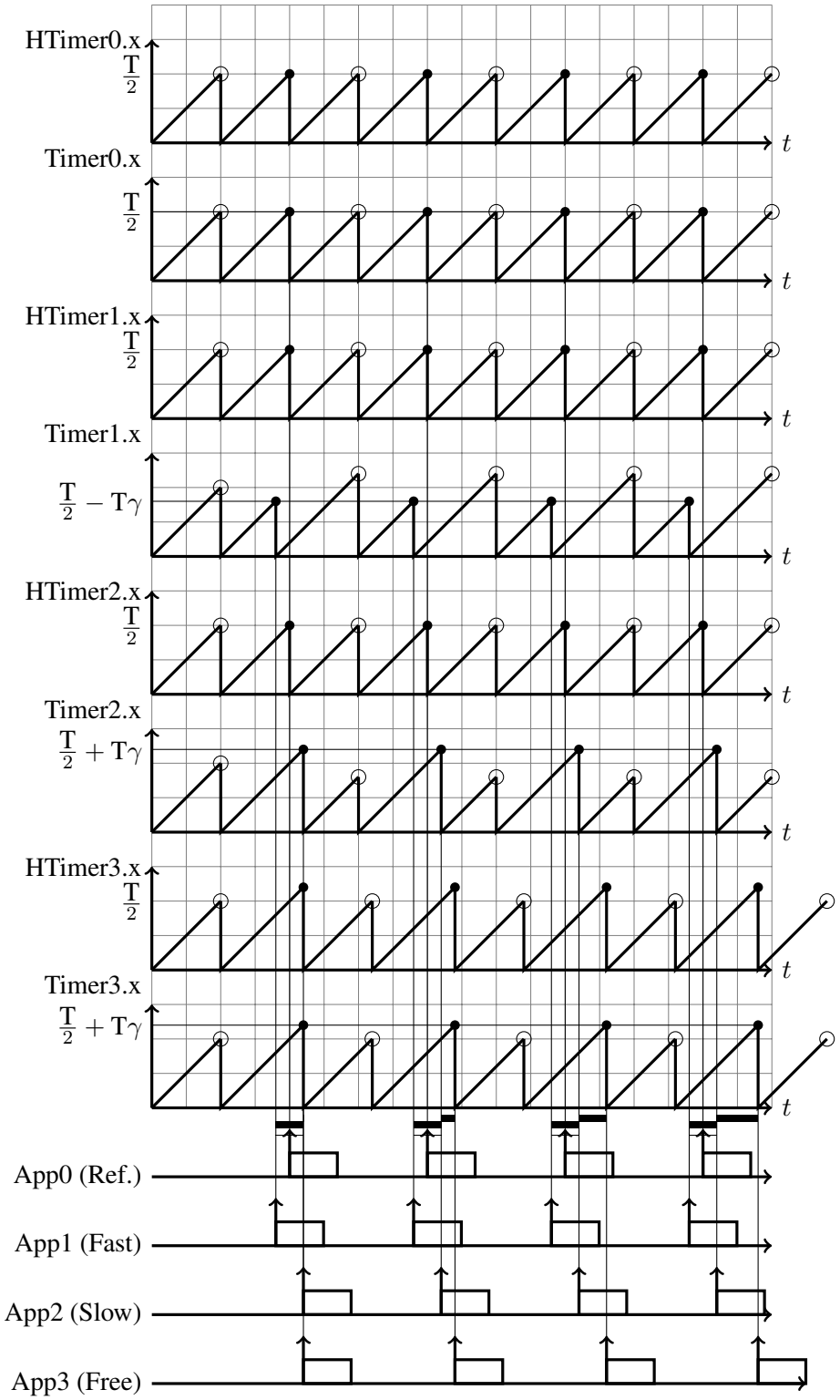


Figure 8.21: Four nodes with virtual clocks, example of an inconsistent clock synchronization

Table 8.2: Results obtained for different values of `maxOD`

`maxOD`	Upper bound (X) [time units]
0	6
1	18
2	30
3	42

between the value of `maxOD` and the upper bound of the offset. This is certainly the expected behavior for the system, as it was already discussed in Section 7.6. Thus, the results show that the modeling pattern realistically models the expected temporal behavior of this kind of systems.

Notice that the obtained values are all multiple of 6. This is due to the fact that the consonance between nodes is always bounded by a discrete value of the form $\gamma_0 k$, with $k \in \mathbb{N}$, and therefore the upper bounds of the offset between nodes can only take values of the form $(2\gamma_0 \mathrm{T})k = (2\mathrm{e}0)k = 6k$.

8.7 Discussion

At the beginning of this chapter, we highlighted the relevance of having suitable modeling patterns for specifying distributed systems with computer clocks. Where *suitable* means having a model that allows us to *realistically* specify the temporal behavior enforced by the type of computer clocks the system works with. Another important characteristic of a suitable modeling pattern is that it makes the assumptions on the computer clocks explicit within the model.

In this chapter we have presented a complete study of the different types of computer clocks that can be encountered when modeling a distributed system, and we have discussed what kind of temporal behavior they enforce. For that discussion, we have defined a simplified case study, constituted by a number of nodes executing a periodical time-triggered task.

For each type of computer clock we have presented and discussed its corresponding modeling pattern. Some of these modeling patterns already existed in the literature, whereas some others were specifically developed for this thesis. The discussed patterns are all summarized next.

Modeling patterns already existent in the literature:

- Ideal timers, for modeling systems with ideal clocks. The temporal behavior of the nodes is perfectly synchronous; they execute the periodical task at exactly the same instants.

- Perturbed timers, for modeling systems with physical clocks. The temporal behavior enforced is unbounded *clock skew*, which means that the execution instants of the nodes keep drifting away as time passes by.

Modeling patterns developed in this thesis:

- Perturbed timers and a clock pointer, for modeling systems with virtual clocks. The temporal behavior enforced is *clock jitter*, which means that the execution instants of the nodes are different, but the offset is still bounded by a certain amount that depends on the clock synchronization algorithm.
- Perturbed timers and multiple clock pointers. The main characteristic of this modeling pattern is that it allows the offset and the consonance of the virtual clocks to change *dynamically*, as a consequence of the actions performed by the nodes during the system operation. Due to this, this pattern allows specification of all the previous types of clocks: ideal clocks, physical clocks and virtual clocks. However, it is especially useful for modeling clock synchronization algorithms.

We also remarked the importance of describing the modeling patterns in a manner that facilitates comprehension and fosters its applicability. For fulfilling this aim, we have extensively used graphs that show the evolution of the TA clocks of each model. In our experience, reasoning about the perception of time by the nodes of a distributed system is not straightforward and probably constitutes the most challenging aspect of the TA theory. We believe that these graphical representations are the best way to introduce the modeling patterns, particularly to readers not familiar with the TA formalism.

Our aim was to provide these modeling patterns in a way such that they can be immediately applied by other system modelers. However, it is clear that each system modeler must still find out if any change is required before adopting a certain modeling pattern for her particular system. For instance, in Section 9.2.6 we will describe how the modeling pattern for clock synchronization (perturbed timers and multiple clock pointers) has been adapted and used for the formal verification of OCS-CAN.

Chapter 9

Model checking of the precision guaranteed by OCS-CAN

This chapter describes the formal verification of OCS-CAN by means of model checking. As indicated in Chapter 3, model checking is a technique that, given a model of a system and a set of properties, automatically checks whether the properties hold for that model or not. One of the main strengths of model checking is that it automatically generates and evaluates *all* the possible states of the system's model. Therefore, it is very useful for evaluating fault-tolerant systems over complex scenarios, as it frees the user from having to find out all the possible scenarios explicitly. This feature will be exploited in the model checking of OCS-CAN.

Due to its complexity, the formal verification of OCS-CAN will be described gradually. This description will be divided into three parts.

Section 9.1 corresponds to the first part, in which we will address a number of preliminary details about the modeling of our system. In this section we will clarify the aim of the formal verification of OCS-CAN, and we will discuss the main abstractions performed in the modeling.

Section 9.2 corresponds to the second part, in which the UPPAAL model of OCS-CAN will be presented and thoroughly described. We will present the network of processes that constitutes the model of OCS-CAN and will discuss each process in detail. We will specifically discuss the most complex aspects of our model of OCS-CAN, which mainly concern the modeling of the virtual clocks and of the clock synchronization operations with the techniques introduced in Section 8.6.

The third and last part of the description will be addressed in Section 9.3. In this section we will

address the formal verification of the discussed model. We will discuss the verification process as well as the results that have been obtained.

9.1 Preliminary remarks about the modeling of OCS-CAN

Figure 9.1 depicts the elements involved in the model checking procedure, as it was discussed in Section 3.2. The model checker, in this case UPPAAL, takes both the model and the properties as inputs and provides an output that may be *yes* or *no*, depending on whether the properties are satisfied or not. Moreover, whenever the output is *no*, the model checker supplies the user with a trace that shows in which conditions the property is violated.

Both the model and the properties are provided by the user, and they must be specified in the formalisms required by the model checker. For the case of UPPAAL, the model must be specified as a network of timed automata (introduced in Section 3.6.1), whereas the properties must be specified as formulas of temporal logic (discussed in Section 3.6.4).

Before describing our modeling in more detail, it is useful to briefly introduce what the "model" and the "properties" represent in the context of the formal verification of OCS-CAN. We will also discuss the main abstractions of our model.

9.1.1 System model

OCS-CAN has been conceived as a subsystem that can be attached to any CAN-based distributed system for immediately providing a high-precision clock synchronization service. The algorithm for clock synchronization of OCS-CAN is based on a master/slave scheme, but includes some mechanisms for managing master redundancy that eliminate the single point of failure that a single master would represent.

For the purpose of formal verification, OCS-CAN will be considered isolated from the rest of the system to which it may be attached, as it was already done in the analysis carried out in Chapter 7. This is possible because the behavior of an OCS-CAN subsystem is not changed by the distributed system to which OCS-CAN is attached. This property is called *orthogonality* and was discussed in Section 6.2.4.

In fact, the orthogonal design of the CU makes the network be the only element shared by OCS-CAN and the rest of the distributed system. Due to this, the only side effect that the rest of the system may have on the operation of OCS-CAN is a potential increment of the Time Message response time, which would be caused by the background traffic. But this potential increment is already considered

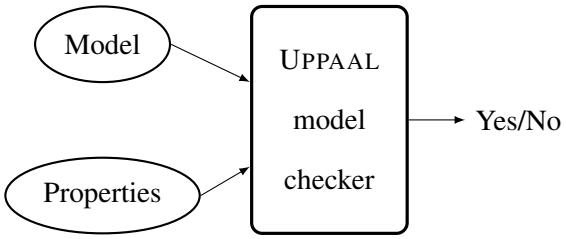


Figure 9.1: Scheme of the model checking procedure

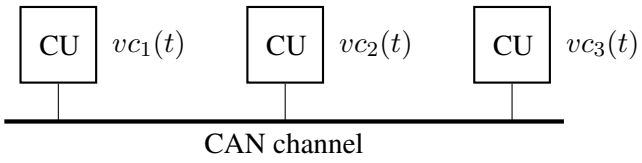


Figure 9.2: An OCS-CAN subsystem made up of three Clock Units

in the formulas for the calculation of the $wcrt$ of each TM, and hence it can be easily incorporated into the model, as it will be shown in Section 9.2.4.

For illustration purposes, Figure 9.2 shows the architecture of an OCS-CAN subsystem made up of three Clock Units, when considered independently. Note that the subsystem can be divided into two parts: the Clock Units (CU) and the CAN channel. Each CU internally keeps a virtual clock for measuring time, labelled $vc_i(t)$, which constitutes the output of the CU to its corresponding node (not represented in the figure).

The internal structure of the CU was described in Section 6.3.1. For the reader's convenience, we have reproduced the block diagram of the CU in Figure 9.3. The blocks that constitute the CU are, namely, the Virtual Clock Module, the Synchronization Module, the Timestamp Manager and the Enhanced CAN Controller. In the following, we will denote them with their abbreviated names, which are respectively: VC module, SynM, TSM and EnCAN.

In order for the formal verification to be complete, the formal model of OCS-CAN must include both elements (the CU and the CAN channel) with their possible behaviors. However, in Section 9.1.3 we will show that certain properties of these elements will not be needed and thus may be abstracted away.

9.1.2 Properties to be verified

The aim of our formal verification is to assess the precision that OCS-CAN guarantees under the different fault assumptions considered in the fault model. The notion of precision was defined in

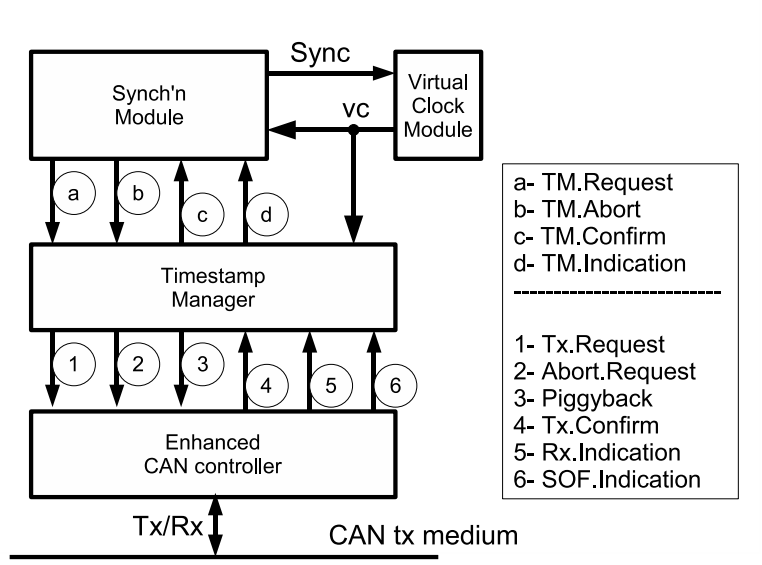


Figure 9.3: Block diagram of the clock unit, with the interface between blocks

Section 7.1.2 (Definition 6) as follows: a set of clock units A is synchronized with precision Π if $|vc_a(t) - vc_b(t)| \leq \Pi$, for any pair of non-faulty clock units $a, b \in A$ and any time instant t. Note that this property can be expressed as a *safety property* of the form: A$\Box$ ($(vc_a - vc_b) \leq \Pi$) $\wedge$ ($(vc_a - vc_b) \geq -\Pi$)).

However, according to this definition of precision, there exists an infinite number of values that may satisfy this property, because if a system is Π-synchronized then it will be also Π'-synchronized for any $\Pi' > \Pi$. Therefore, the aim of our formal verification can be redefined as finding the *minimum value* that satisfies said property. Moreover, due to the limitations of model checking to deal with non-integer numbers, we will hereafter consider that Π is a natural number.

In order to determine this minimum value for the precision, in the model checking of OCS-CAN we will follow a procedure that is divided in the following three steps.

1. The general parameters of OCS-CAN will be set in the model. These parameters are: the length of the synchronization period (R), the number of masters, the value of the release delay (Δ_m), etc.

2. The fault assumptions of the model will be set. These assumptions are: the maximum number of crashed masters, the omission degree (O_{max}), etc.

3. The precision property will be verified iteratively. The iteration starts by checking the precision property for Π=1. If the property is not satisfied then the value of Π is increased by 1, and the

precision property is checked again. Once the precision property is satisfied, the current value of Π constitutes the minimum value of the precision for the given configuration parameters and fault assumptions, and hence we finish the iterative process.

In Section 9.3, after the modeling of OCS-CAN has been thoroughly described, we will provide further details about the specific way to implement this iterative procedure in UPPAAL.

The precision of OCS-CAN must be assessed for different system parameters and fault assumptions. Then, whenever one or more of the system parameters change, the user has to restart the procedure from step 1.

9.1.3 Main abstractions of the model

The first task to be addressed during the formal modeling of any system is to decide which aspects of the system are relevant for the formal verification and *must* be included in the model, and which aspects are irrelevant and *can* be abstracted away. The relevance or irrelevance should be decided in regard to the properties to be verified.

Furthermore, even for those aspects that are relevant, the modeler must consider with what level of detail they have to be modeled. Very often it is enough to just model the properties that a certain component exhibits, instead of explicitly modeling the low-level details of the particular mechanisms that substantiate said properties. Assuming some properties as granted reduces the complexity of the formal model and the size of the state space to be checked.

In order to determine the required level of modeling detail for a certain property, it turns out to be very useful to state what aspects of the system *are not* the goals of the formal verification. This helps the modeler to find out the relevance (or more exactly, the irrelevance) of including certain details in the model.

Concerning the formal verification of OCS-CAN, we can highlight three aspects that will not be addressed:

1. *We will not formally verify the CAN protocol itself.* The CAN protocol is an international standard since 1993 [ISO93]. Its properties have been studied for a couple of decades and are very well known.

2. *We will not formally verify the implementation of the equations for clock amortization.* Model checking is not a suitable technique for formal verification of so-called *data-intensive applications*, i.e. applications that perform a lot of arithmetical operations [Kat98]. When modeled as finite state automata, arithmetical operations usually create huge state spaces because they

force the modeler to include long variables in the model, such as integers, long integers or floats, which can take values in a very wide range. Data-intensive applications are better assessed by means of theorem proving or numerical calculus.

3. *We will not formally verify the internal fault tolerance mechanisms of the CU*, which are intended to restrict the failure semantics of the CU to *crash failure semantics*. The design of these mechanisms has not been addressed in this work and hence there is no particular mechanism to be formally verified. Therefore, we will assume that each CU exhibits this property without considering further details.

As a consequence of these points, in the modeling of OCS-CAN we will make the following abstractions:

- The CAN network will be modeled as a "black box" that provides a broadcast service fulfilling the main properties of CAN. The way to model such a black box by means of timed automata will be thoroughly discussed in Section 9.2.4.
- Instead of the equations for clock amortization, our model will assume immediate clock assignment. Thus, we adopt the same strategy applied in Section 7.2.1.
- The modeling of the CU will include a state called *failure*, which will be used for modeling the crash of the CU. Any CU will be able to indeterministically enter this state .

9.2 Description of the UPPAAL model of OCS-CAN

This section introduces the basic scheme of the model, as a network of UPPAAL processes, and describes the most simple features of these processes.

9.2.1 Basic scheme of the UPPAAL model

The scheme of the formal model of OCS-CAN is depicted in Figure 9.4. Every rounded rectangle in the scheme represents an UPPAAL process. An arrow indicates some kind of communication between two processes, usually by means of an UPPAAL channel or a global variable.

The model of OCS-CAN includes two processes for modeling each CU, `SynM` and `VC module`, and one process for modeling the CAN communication, `Channel`. The latter corresponds to the "black box" mentioned in Section 9.1.3 for modeling the CAN network. There are also two other

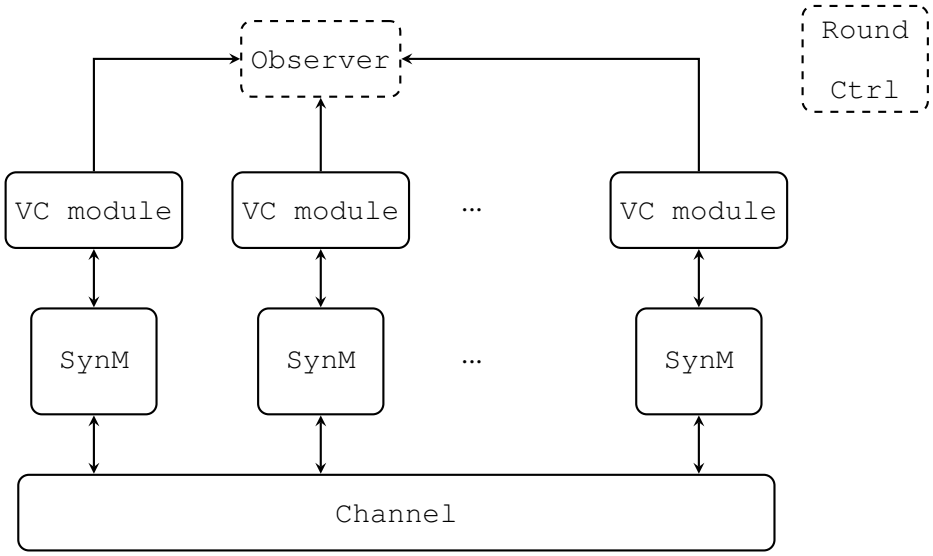


Figure 9.4: General scheme of the formal model of OCS-CAN

processes, `Observer` and `Round Ctrl`, both remarked with a dashed line in Figure 9.4, whose function will be clarified later on in this section.

Note that no specific process is defined for modeling the behavior of the blocks TSM and EnCAN. This is a consequence of the abstractions discussed in Section 9.1.3. Modeling the functionality of EnCAN is not required since it mainly deals with the low-level details of the CAN protocol. For this reason, EnCAN is abstracted away as part of process `Channel`. The timestamp functionality of TSM is abstracted away as part of process `SynM`.

The rest of this section is devoted to describing the processes that constitute the core of the model: `VC module`, `SynM` and `Channel`.

9.2.2 The process `VC module`

The function of `VC module` is to model the temporal evolution of the virtual clock. This is achieved by means of two variables: one that represents the value of the virtual clock and another one that represents the speed (or rate) of the virtual clock. We adopt the notation of Chapter 7, so that these variables are called, respectively, vc_a and $\dot{vc}_a$ hereafter.

The normal behavior of vc_a is to increase monotonically over time, at the pace indicated by $\dot{vc}_a$. Nevertheless, vc_a can also change its value abruptly at a certain synchronization instant, as a consequence of the *offset adjustment* that was discussed in Section 7.2.1. The value of $\dot{vc}_a$ may also change at a synchronization instant, as a consequence of a *drift adjustment*, but remains constant between

said synchronization instants.

The modeling of this apparently simple behavior constitutes by far the most challenging aspect of the OCS-CAN model. The main difficulty is related to the fact that all the clocks in a network of timed automata increase, by definition, at the same pace. Therefore, a virtual clock of OCS-CAN cannot be directly modeled as one of such clocks; more complex modeling techniques are required.

This circumstance forced us to develop and apply some complex modeling patterns, not only to specify virtual clocks with different rates, but also to specify virtual clocks whose rates may change according to the rate of another virtual clock (the master). These modeling patterns were thoroughly discussed in Chapter 8.

For the modeling of `VC module` we will use a variation of the modeling pattern proposed in Section 8.6.2 for clock synchronization. However, a full description of the complex aspects of `VC module` is not really required for understanding the basic features of the model, which we are addressing in this section. Therefore, and for the sake of clarity, this description will be postponed until the last part of the description, in Section 9.2.6.

9.2.3 The process `SynM`

Process `SynM` models the behavior of the Synchronization Module of the CU, which is based on the algorithms described in Section 6.3.4. The algorithm executed by this module may change, depending on whether `SynM` plays the role of a master or of a slave. The algorithm for a master clock unit is reproduced in Figure 9.5, whereas the algorithm for a slave clock unit is reproduced in Figure 9.6.

These two algorithms exhibit a slight difference with respect to the algorithms discussed in Section 6.3.4. In the automata of Figure 9.5 and Figure 9.6, the synchronization action has been abstracted away and has been substituted by an *immediate clock assignment*. This substitution can be observed in the transitions of the master from `Queue` to `Idle 2` and from `Idle 1` to `Idle 2`, when the event TM.Ind$(n) \wedge n \in hp(m)$ is detected, what indicates that a higher-priority TM has been received. It can be observed in the slave as well, in the transition from state `Idle 1` to state `Idle 2` specifically.

In essence, we have replaced the synchronization action by the effect that the action would have on the virtual clock. This is a valid abstraction because, as indicated in Section 9.1.3, we are not interested in model checking the clock adjustment operations themselves.

It is also important to remark that this way of modeling the clock adjustment implicitly assumes that the value and the rate of the master's virtual clock are available to any CU. This represents an abstraction of the timestamp mechanism implemented between TSM and EnCAN. As it was explained

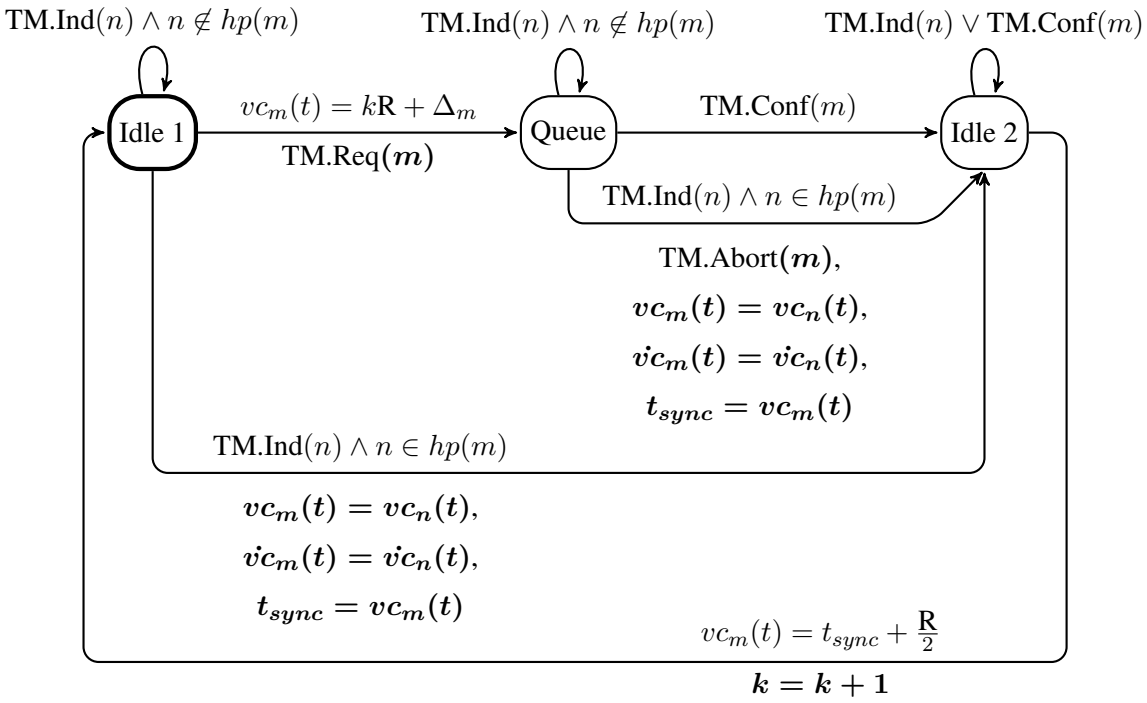


Figure 9.5: Algorithm executed by the SynM of master m, with the Sync(n, m) operation abstracted away

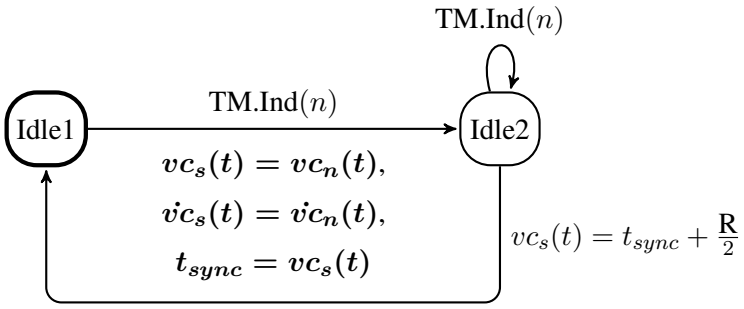


Figure 9.6: Algorithm executed by the SynM of slave s, with the Sync(n, s) operation abstracted away

in Section 6.3.2, the value of the master's virtual clock is piggybacked into the data field of the TM by the master, and thanks to this (and after reception of the TM) it is available to the other CU. But the inclusion of the timestamp mechanism is not required in the modeling of OCS-CAN, and we assume instead that the value (vc_n) and the rate ($\dot{vc}_n$) kept by the `VC module` of master n are global variables that can be read by any other process.

The way of modeling the error caused by the timestamp, and the effect it may have on the adjustment of a virtual clock belongs to the modeling of `VC module`, and therefore will be further discussed in Section 9.2.6.

Concerning the interaction between process `SynM` and process `Channel`, our model includes the four primitives presented in Section 6.3.1, namely TM.Req(m), TM.Conf(m), TM.Ind(n) and TM.Abort(m). The modeling of these primitives with timed automata will be thoroughly discussed in Section 9.2.4, while describing process `Channel`.

As a final remark, note that the description of `SynM` given so far is incomplete, since we have not indicated how these algorithms are modeled with timed automata. The timed automaton of process `SynM` will be partially introduced in Section 9.2.4, along with the timed automaton of process `Channel`. Nevertheless, the final (and complete) timed automaton of `SynM` will be presented in Section 9.2.6.

9.2.4 The process `Channel`

The function of process `Channel` is to model the communication services provided by the CAN network. This model must satisfy the properties of CAN discussed in Section 4.3, such as arbitration and bounded response time. But, more importantly, and given that our aim is to assess the precision guaranteed by OCS-CAN *in the presence of faults*, this process should include all the possible failure semantics of the CAN network.

The error-control mechanisms of CAN guarantee, in the presence of most channel faults, the consistent broadcast of any message transmitted. However, in some specific fault scenarios, the service provided by a CAN network may fail and cause either Inconsistent Message Duplicates (IMD) or Inconsistent Message Omissions (IMO). In Section 7.4.1 we already indicated that the possible failure semantics of a CAN network are: Consistent Broadcast (or **Br-C**), Broadcast with inconsistent message duplicates (or **Br-ID**) and Broadcast with inconsistent message omissions (or **Br-IO**).

In principle, these failure semantics should be all included in the modeling of OCS-CAN. Nevertheless, in the analysis carried out in Section 7.4.3, it was proved that, from the perspective of the guaranteed precision, having Br-ID is equivalent to having Br-C. Due to this, and in order to reduce the complexity of the model, the possibility of Br-ID will not be included in our modeling. This constitutes an important abstraction, but it can be safely done as it has no impact on the property to be verified.

In summary, process `Channel` will model a broadcast service fulfilling these properties:

- **Inconsistent broadcast of messages**. It is possible that a subset of the CU receive a certain

message whereas the other CU do not receive it (due to an IMO). In consonance with the nomenclature defined in Section 7.4.4, the maximum number of rounds that may suffer from an inconsistency will be called the *maximum omission degree* and will be denoted as O_{max}.

- **Simultaneous reception of messages**. Given that IMD have been excluded from the modeling, we assume that all of the CU that receive a message, receive it simultaneously.
- **CAN arbitration**. Whenever two CU (two masters, actually, since slaves are not allowed to send any TM) request the broadcast of a message at the same time, the message with higher priority is broadcast first.
- **Bounded response time**. Any message broadcast is received before $wcrt_m$ time units or it is not received at all. The calculation of $wcrt_m$, which is to be performed offline, must include the delay caused by potential channel errors and retransmissions. The response time of a TM has also a lower bound, which is denoted as $bcrt_m$.
- **Broadcast abortion**. Any message broadcast may be aborted by the CU that requested the broadcast.

In order to specify this semantics, process `Channel` will use the following elements:

- A local clock `x`, which is used for implementing the bounded response time of CAN.
- A global variable, `msg_id`, and an urgent channel, `tx_req` for modeling the broadcast requests, including the arbitration.
- A global variable, `recv_id`, and a broadcast channel, `tx_msg`, for modeling the reception of the TM.

It is important to remark that process `Channel` interacts with process `SynM` for modeling the broadcast and reception of the TM, including the arbitration, and with process `Round Ctrl` for modeling message inconsistencies. These interactions will be clarified in the following discussions.

Modeling TM broadcast and arbitration

The automaton that specifies the behavior of process `Channel` is depicted in Figure 9.7. Note that this automaton has only two locations. The one named `no_pending_tx` corresponds to the state in which no TM broadcast has been requested, and is thus the initial state; whereas the location named `pending_tx` corresponds to the state in which at least one TM is waiting for transmission. The initial values of the global variables `msg_id` and `recv_id` are N and 0, respectively.

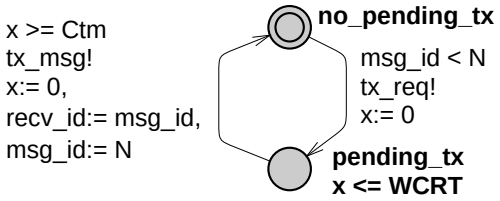


Figure 9.7: Automaton of process `Channel`

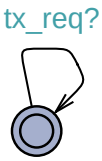


Figure 9.8: Dummy automaton for enabling synchronization via the urgent channel `tx_req`

The transition to `pending_tx` is guarded by the condition `msg_id < N` and has a synchronization through the urgent channel `tx_req`. We use a so-called *dummy automaton* (depicted in Figure 9.8) in order to make the synchronization via channel `tx_req` always possible. This strategy guarantees that the transition is taken as soon as the guard condition becomes true, which should occur only when a master has a TM to broadcast.

The value of `msg_id` (N initially) may be modified by a `SynM` that wishes to transmit a TM. Therefore, overwriting the value of `msg_id` is equivalent to requesting a broadcast of the TM. In fact, it corresponds to the primitive TM.Req(m), where m is the value that `msg_id` will take. However, in order to guarantee the arbitration property of CAN, a `SynM` requesting a broadcast will overwrite the value of `msg_id` only if the identifier of its own TM is lower than the current value of `msg_id` (since a lower identifier means higher priority). The way to include this in `SynM` is shown in Figure 9.9. Note that this figure does not show a complete automaton, but only the "piece" that illustrates the mechanism for modeling TM.Req(m) and arbitration.

In the automaton of Figure 9.9, `my_id` is a local constant that corresponds to the identifier of the TM sent by the node, so it is actually the value of m. The expression `msg_id := my_id <? msg_id`, which UPPAAL has taken from the C language, is a compact way to write the code appearing in Algorithm 3.

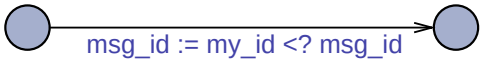


Figure 9.9: Portion of an automaton that models TM.Req(m) and arbitration

Algorithm 3 The simple *if-else* statement that serves for modeling CAN arbitration

if my_id $<$ msg_id **then**

  msg_id $\leftarrow$ my_id

else

  msg_id $\leftarrow$ msg_id

end if

Modeling the channel's bounded response time

Process `Channel` also models the bounded response time of the CAN network. As previously indicated, the response time of a certain TM should always lay within the interval $[bcrt_m, wcrt_m]$. Both values are calculated offline and included into the model as constants.

The lower bound ($bcrt_m$) is calculated as follows: let L_{tm} be the length of the TM (measured in bits) and let C be the bit rate of the channel (measured in bps), then $bcrt_m = L_{tm}/C$. Since the length of the TM is fixed, the value of $bcrt_m$ is the same for any TM.

It also is possible to calculate the value of $wcrt_m$ for the TM issued by each master, for instance as indicated in [BBRN05, DBBL07]. But in our model we will assume the same $wcrt_m$ for every TM. This is a valid abstraction because the duration of $wcrt_m$ is negligible in comparison to the synchronization period R. And given that the length of R is the factor that more significantly affects the precision, the error that this abstraction may induce on the assessment of the precision is very little. Moreover, it is possible to apply a conservative approach: if the value assigned to the upper bound corresponds to the greatest $wcrt_m$ among the TM, this causes pessimism on the assessment of the precision. Therefore the precision obtained as a result of the formal verification is still an upper bound and therefore valid.

The way of introducing in our model the upper and lower bounds of the response time can be observed in the automaton of Figure 9.7. Note that clock `x` is reset in the transition to `pending_tx`. This clock is then used for restricting the time that the automaton can stay in location `pending_tx`, which, due to the guard and the invariant defined over clock `x`, lays between `Ctm` and `WCRT`. The constant `Ctm` corresponds to $bcrt_m$, whereas the constant `WCRT` corresponds to $wcrt_m$.

The transition from `pending_tx` to `no_pending_tx` indicates a successful broadcast of the

TM. This event is notified through the broadcast channel `tx_msg`. In the same transition, process `Channel` overwrites the value of the global variable `recv_id` with the value of `msg_id` and resets the value of `msg_id` to N, which makes the channel available again for other broadcasts.

Modeling TM indication and TM confirm

As already indicated, the broadcast of the TM takes place in the transition from `pending_tx` to `no_pending_tx`, and it is signaled through the channel `tx_msg`. Channel `tx_msg` is a broadcast channel, so that (according to what was explained in Section 3.6.2) multiple processes may use it for synchronization. Therefore, this single signaling will be used for modeling the TM.Ind(n) at the (multiple) receiving `SynM` and the TM.Conf(m) at the (single) transmitting `SynM`.

The specific way of modeling these primitives is depicted in Figure 9.10. This automaton is a simplification of the final automaton of a `SynM` playing the role of a master, and is incomplete. But, despite its simplicity, this automaton is very useful for understanding the communication between each process `SynM` and process `Channel`.

Before describing the simplified automaton of Figure 9.10, it is important to remind that right before a TM broadcast is signaled, the global variable `msg_id` keeps the identifier of the highest-priority TM among those that are waiting for transmission. So this variable gives the identifier of the TM that is actually broadcast. Therefore, every process `SynM` can check this variable in order to know whether the TM that has been broadcast is its own TM or it is a TM broadcast by another process. Moreover, if the latter is true then it is also possible to discriminate between a TM sent by a master of higher priority and a TM sent by a master of lower priority.

The automaton of Figure 9.10 contains three locations, named `Idle1`, `Queue` and `Idle2`. Its behavior corresponds to the algorithm discussed in Section 9.2.3, even though with a few simplifications. Note that while being in the initial location `Idle1`, three events may happen:

1. The reception of a lower priority TM may be signaled via `tx_msg`. This transition is the one guarded by the condition `msg_id > my_id`. Note that it does not cause a change of location because, according to the algorithm discussed in Section 9.2.3, each lower priority TM is basically ignored.

2. The reception of a higher priority TM may be signaled via `tx_msg`. This transition is the one guarded by the condition `msg_id < my_id`. In a complete automaton of `SynM`, this transition should cause a synchronization of the virtual clock. In our simplistic model, it makes the automaton change to location `Idle2`.

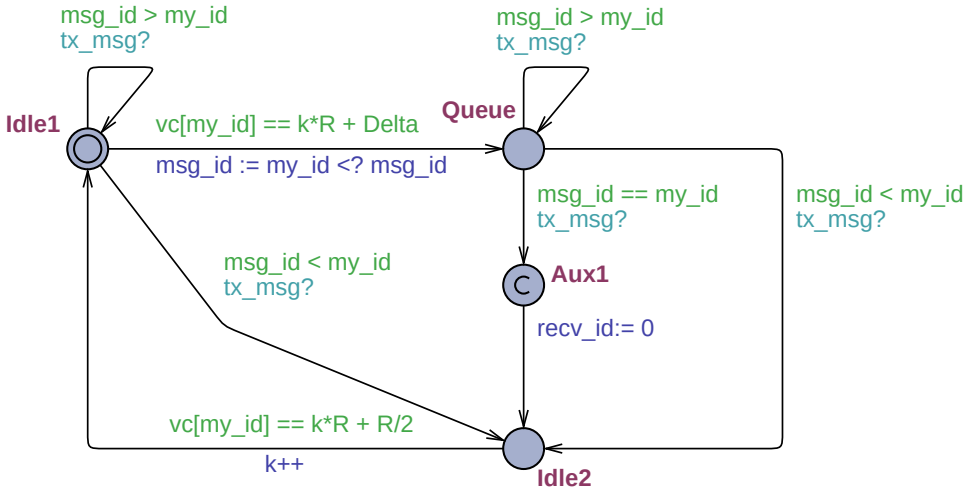


Figure 9.10: Simplified automaton of a master `SynM` (version I)

3. The virtual clock may reach the so-called *broadcast instant* of SynM. In such a case, a broadcast of a TM with identifier `my_id` will be requested in the form previously explained; i.e. by overwriting the value of `msg_id`, if possible. Note that in the transition from `Idle1` to `Queue`, the *update* corresponds exactly to the one depicted in Figure 9.9, which modeled the TM Request.

While being in location `Queue`, three events are possible:

1. An indication of a TM of lower priority.
2. An indication of a TM of higher priority.
3. A confirmation that the broadcast previously requested has been successful. This corresponds to the TM.Conf(m) primitive, and it is modeled in a similar way to TM.Ind(n); by means of a transition that synchronizes via channel `tx_msg`, but guarded by the condition `msg_id == my_id`.

Note that the transition caused by TM.Conf(m) leads to a committed location, named `Aux1`, which is therefore immediately left. In the subsequent transition, to location `Idle2`, the global variable `recv_id` is reset to 0. This action guarantees that variable `recv_id` takes the value 0 as soon as it is not required anymore, and helps to reduce the state space. The utility of variable `recv_id` is specifically related to the modeling of the clock adjustment operations and has not been explained yet, but it will be clarified in Section 9.2.6.

It is very important to remark that the transitions that model the reception of a *lower* priority TM (i.e. the events with number 1 in the lists above) have been shown in Figure 9.10 for completeness, but they will be abstracted away from our model hereafter. Such transitions return to the same location and do not modify any clock or variable, which means that they do not cause any change of state of the process. Therefore, they can be safely eliminated without modifying the state space to be checked.

Modeling TM abort

Due to the particular mechanism we have used for modeling both the requests of TM broadcasts and the arbitration, there is no need to include any mechanism to model abortion of the TM. In our model, any TM broadcast is aborted in practice, as soon as any other process `SynM` overwrites the value of `msg_id` with an identifier of higher priority (i.e., of lower value). For this reason, what we need to include is the opposite mechanism: a mechanism to re-request a TM broadcast whenever the abortion of a TM has not taken place.

In Section 6.3.4 we explained that a TM may not be aborted while it is being transmitted, and that due to this, the TM.Abort(m) primitive may be unsuccessful sometimes. This possibility is included in our model by means of the modeling technique shown in Figure 9.11. Note that in this new version of process `SynM`, three new locations appear in the right path from `Queue` to `Idle2`. This path corresponds to the reception of a higher priority TM after a broadcast of the TM has been requested.

Location `Abort` is a committed location and is immediately left, but it gives us the possibility of specifying the two possible outputs of the TM.Abort(m) primitive: successful abortion of the TM and unsuccessful abortion of the TM. The successful abortion is represented by the unguarded transition to location `Idle2`. This transition does not need any specific action, since the previously requested broadcast is aborted by default.

The unsuccessful abortion of the TM is represented by the unguarded transition from `Abort` to `Not_aborted`. Notice that in this transition, `SynM` requests a new broadcast of the TM. Location `Not_aborted` is left once there is either an indication of the reception of a higher priority TM (guard `msg_id < my_id`) or a TM confirm (guard `msg_id == my_id`). Location `Aux2` is just an auxiliary committed location for resetting the value of `recv_id` to 0.

Modeling omissions of the TM

For modeling omissions of the TM, we must allow each `SynM` to act as if the TM indication had not been issued by process `Channel`. For including this possibility, we will define an extra transition in `SynM` that also synchronizes via the broadcast channel `tx_msg` and is enabled at the same time as the

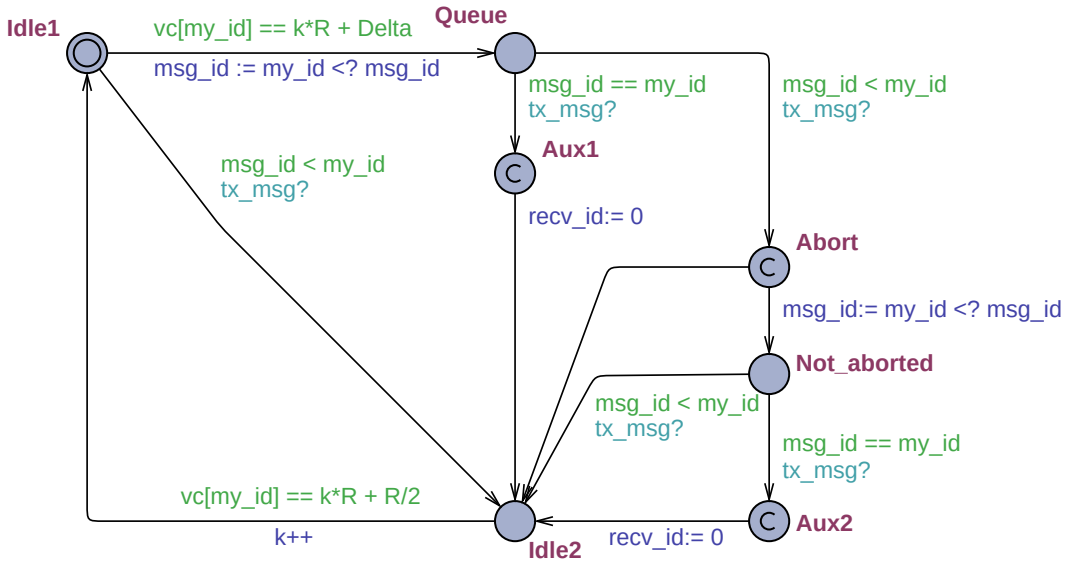


Figure 9.11: Simplified automaton of a master `SynM` (version II)

transition corresponding to the TM indication. We call this transition an *omission transition* because whenever it is taken, it prevents process `SynM` from processing a TM indication.

Nevertheless, an important assumption of our model is that the number of consecutive inconsistent rounds is bounded by the maximum omission degree (O_{max}). Therefore, the omission transition should be enabled/disabled according to this value. We define a boolean global variable, called `eomission`, for this purpose. The value of this boolean variable indicates whether the current synchronization round may suffer from inconsistencies (TRUE) or not (FALSE). Due to this, `eomission` must be set to FALSE after O_{max} consecutive inconsistent rounds in order to force a consistent synchronization round, and it must be set to TRUE otherwise.

The automaton of Figure 9.12 corresponds to a new version of a master `SynM`, which illustrates the use of our technique for modeling TM omissions. This automaton incorporates one omission transition in location `Idle1` and another one in location `Queue`. These transitions are both guarded by the condition `eomission and (msg_id < my_id)`, what means that they are enabled only during potentially inconsistent synchronization rounds and for higher priority TM.

Each one of the omission transitions leads to a committed location (`omission0` and `omission1`, respectively) that is immediately left. Furthermore, in the transition leading from `omission1` to `Queue`, a transmission request of the TM is also performed. This action allows us to specify that an omission of the TM will actually prevent `SynM` from aborting the broadcast previously requested.

In Figure 9.12, it must be noticed that whenever one of the omission transitions is taken, a boolean

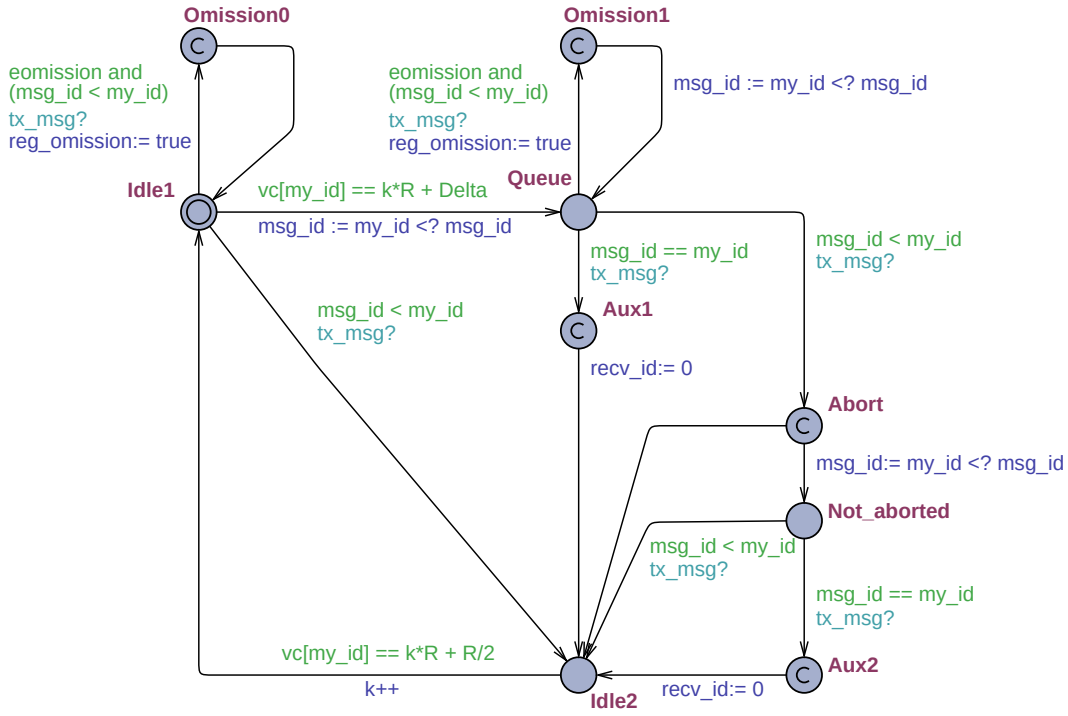


Figure 9.12: Simplified automaton of a master `SynM` (version III)

global variable called `reg_omission` is set to TRUE. This variable is used by another process (process `Round Ctrl`) for counting the number of consecutive inconsistent synchronization rounds and setting the value of `eomission` accordingly.

The control of the number of consecutive inconsistent rounds is performed by process `Round Ctrl`. As already indicated, the main function of this process is to enable/disable the possibility of TM omissions by means of the global variable `eomission`.

The timed automaton of process `Round Ctrl` is shown in Figure 9.13. Notice that this automaton has only one location and one transition. This transition is fired as soon as the guard condition `num_clk == N` is satisfied, because channel `all_end_round` is also an *urgent* UPPAAL channel. The guard condition indicates that each one of the N clock units has finished its synchronization round. Thus, this guarantees that variable `eomission` is only reevaluated at the end of each synchronization round.

The evaluation of variable `eomission` is performed within the *update* associated to the transition, as it can be seen in Figure 9.13. The procedure corresponds to Algorithm 4, and it is divided into three steps.

In the first step, the current number of consecutive inconsistent rounds (kept in the global integer

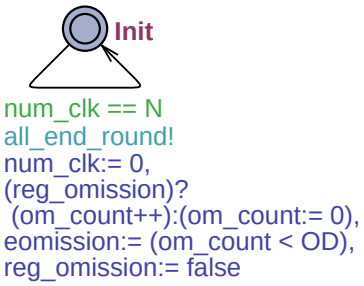


Figure 9.13: Automaton of process `Round Ctrl` (version I)

variable `om_count`) is updated depending on the value of the global variable `reg_omission`. If the value of `reg_omission` is TRUE, it means that in the last synchronization round there was at least one omission of the TM, and therefore the value of `om_count` is increased in one unit. In contrast, if the value is FALSE then variable `om_count` is set to 0, meaning that the count of consecutive inconsistent synchronization rounds is restarted.

In the second step of the procedure, the value of `om_count` is compared to O_{max}, which in the automaton is represented with the constant `OD`. If `om_count` has not reached O_{max} then the next synchronization round may be inconsistent, so that variable `eomission` is set to TRUE. But once `om_count` has reached O_{max}, inconsistencies must not be allowed for the next synchronization round and hence `eomission` is set to FALSE.

In the third and last step of the procedure, the variable `reg_omission` is set to FALSE again. This allows each process `SynM` to report any omission of the TM that may happen in the next synchronization round.

Algorithm 4 Algorithm for managing inconsistencies, as executed by process `Round Ctrl`

if reg_omission = TRUE **then**

  om_count ← om_count + 1

else

  om_count ← 0

end if

eomission ← (om_count < OD)

reg_omission ← FALSE

It is important to highlight that, as long as `eomission` is TRUE, the reception or omission of a TM is decided by every process `SynM` independently from the actions carried out by the other `SynM`. This

implies that any combination of message inconsistencies can occur in our model: from the consistent reception of the TM to the consistent omission of the TM, any other combination is possible. For instance, all of the complex scenarios that where analyzed in Section 7.6 are automatically generated by the model checker.

The automatic generation and evaluation of all possible scenarios is one of the most powerful features of model checking. It guarantees that the worst case scenario is always considered during the model checking process and, therefore, the properties verified do hold under any condition considered within the system model. This can be compared to other techniques, such as testing, in which it is impossible to create and measure every possible fault scenario. Even with simulation, very often it is not possible to assess every potential scenario and then the validity of the obtained results must be characterized with some coverage. For this reason, it is sometimes said that model checkers are the ultimate simulation tool, as they provide 100% coverage [Kat98].

9.2.5 Modeling internal faults of the CU

As indicated in Section 7.1.1, OCS-CAN is made up of a set of CU interconnected through a CAN channel. The CU is constituted by four blocks: VC module, SynM, TSM and EnCAN. In principle, the fault model of OCS-CAN accepts that any of these four blocks may fail, but it assumes that there is some kind of internal fault tolerance mechanism that guarantees that any internal fault manifests as a stop of the whole CU. Whenever a system fulfills this property, it is said to exhibit *crash failure semantics*.

A common way to substantiate the crash failure semantics is by means of internal duplication and comparison, but the adoption of any specific technique is out of the scope of this work and has not been addressed yet. Moreover, knowing which particular internal fault tolerance mechanism has been adopted is not required for the formal verification of OCS-CAN. As discussed in Section 9.1.3, there is no need to model the internal details of certain mechanisms, but only the properties they enforce.

In this section we will discuss how to incorporate the possibility of internal faults of the CU in the modeling of OCS-CAN. We will show that this feature can be incorporated without introducing too much complexity, thanks to the following reasoning:

i. The initial assumption is that any internal fault of a CU manifests as a stop (or crash) of the CU.

ii. Among the actions performed by a CU, the only one that is visible by the rest of the CU is the broadcast of a TM. Then, the crash of a CU can be perceived by the rest of the system only as an omission to send the TM.

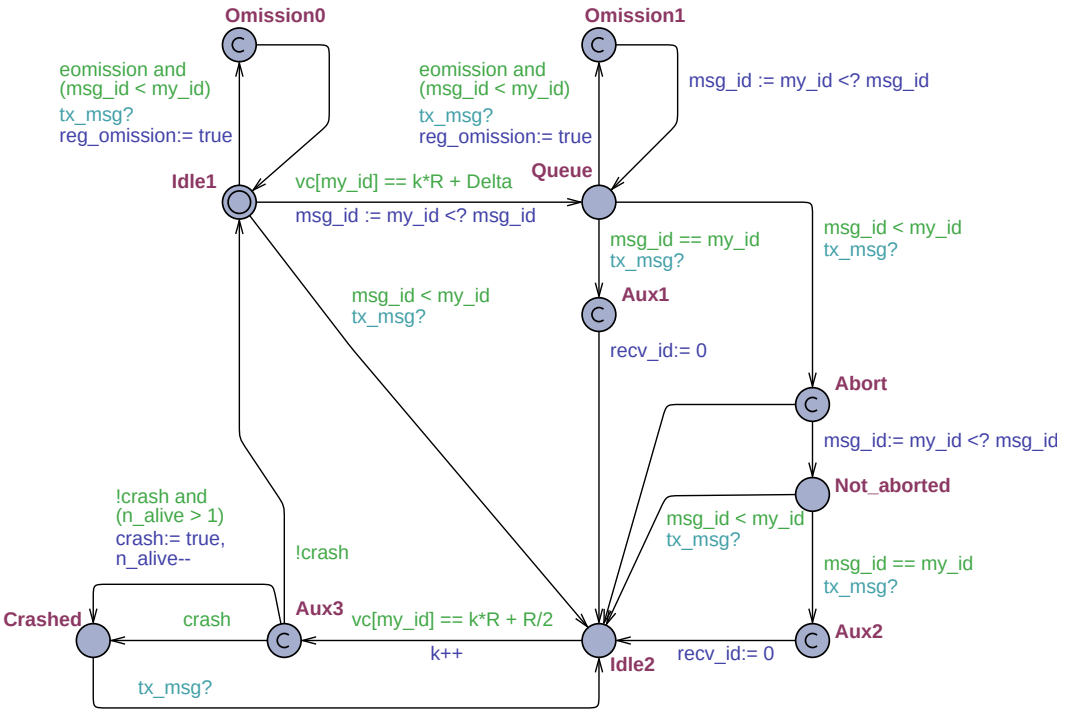


Figure 9.14: Simplified automaton of a master `SynM` (version IV)

iii. Only masters may send a TM. Therefore, from the perspective of the other CU in the system, there is no difference between the behavior of a non-faulty slave and the behavior of a faulty slave. Due to this, and given that the precision is defined only for non-faulty clock units, the crash of a slave will not have any effect on the guaranteed precision. Then, there is no need to include the possibility of crash in our model of the slaves.

iv. The broadcast of the TM is triggered by SynM. Therefore, allowing SynM to enter a permanent state in which the TM is not broadcast anymore would be enough for modeling a master suffering a crash failure.

Another important fault assumption of OCS-CAN is that there is always at least one non-faulty master clock unit in the system. For this reason, the model should keep a count of the number of non-faulty masters in the system and disable the crash possibility when all but one masters are faulty.

Figure 9.14 shows a new version of a master `SynM`, which includes the modeling of crash failures. Notice that in this automaton we have placed a committed location, called `Aux3`, at the end of the synchronization round, between locations `Idle1` and `Idle2`. This new location introduces the possibility of having a crash in the *next* synchronization round, as explained next.

On the one hand, the transition from `Aux3` to `Idle1` corresponds to the behavior of a non-faulty master. This transition is guarded by the condition `!crash`, where `crash` is a local boolean variable that is initially set to FALSE and is set to TRUE as soon as the faulty state is entered.

On the other hand, the transitions from location `Aux3` to `Crashed` correspond to the behavior of a faulty master. The upper transition, guarded by the condition `!crash and (n_alive > 1)`, can be taken only once and indicates that a crash has happened. The consequence of this transition is that the local variable `crash` is set to TRUE and that the global integer variable `n_alive` is decreased in one unit. Variable `n_alive` is initially set to N (the total number of masters in the system), and indicates the number of non-faulty masters. When its value reaches 1, no more crashes are allowed.

Note that once the value of `crash` is set to TRUE, location `Idle1` cannot be reached anymore, because the transition leading to `Crashed` is the only one enabled from `Aux3`. This guarantees that a faulty master may not broadcast its TM again.

Although the transition from `Crashed` to `Idle3` seems useless at this point of the description of process `SynM`, it is required for properly modeling the virtual clocks. As it was discussed in Section 8.2.3, having a TA clock whose value increases indefinitely is a potential risk for model checking because it may increase the state space indefinitely as well. Thus, in order to reduce the state space, each virtual clock must be restarted in every synchronization round. The function of the mentioned transition is to allow this restarting.

9.2.6 The final model

In this section, we will present and describe the final UPPAAL model of OCS-CAN. This final model integrates what has been described throughout Section 9.2 with the modeling pattern for clock synchronization discussed in Section 8.6. After that, we will also discuss the specific procedure we have followed for model checking the achievable precision, and we will present some of the results obtained under different fault assumptions.

In Section 9.2, we discussed the basic aspects of our model of OCS-CAN. In the following subsections we will discuss the most advanced aspects of the model, mainly those that concern the modeling of the virtual clocks. We will indeed describe the final version of the processes `VC module` and `SynM`, which could not be fully described in Section 9.2.

Final model of process `VC module`

It is important to remark that the functionality of `VC module` changes slightly if the process is related to a master node or to a slave node. Due to this, we will define two different types of processes

VC module, with slight variations between them. They are both explained next.

Figure 9.15 depicts the timed automaton of VC module for a master. This automaton has four locations. Three of them, namely Sync_first_half, Wait_delta and Sync_second_half correspond to the three relevant phases of every synchronization round, from the master's perspective. The last location, with no name, is used only for coordinating the different processes VC module so that they step into location Sync_first_half simultaneously.

This automaton uses a global variable, R1 that equals $\frac{R}{2}$. This value corresponds to half the length of the synchronization period, and allows us to use the concept of Half timer discussed in Section 8.5, although with some changes. The automaton also uses a global variable Delta[clock_id] that keeps the value of the master's *release delay* (Δ_m).

Note that while being in location Sync_first_half, process VC module behaves as a perturbed automaton that measures a duration of R1 over clock vc[clock_id], with a variation that depends on the consonance of vc[clock_id] with respect to the reference. This variation is kept in variable eps[clock_id], which means that the transition occurs at a certain instant that lays within the range [R1-eps[clock_id], R1+eps[clock_id]].

As soon as the transition to Wait_delta is fired, a local clock named aux_clk is restarted and a global variable named n_sync is increased in one unit. As it will be explained in Section 9.3, variable n_sync is used by the process Observer for assessing the precision. The auxiliary clock aux_clk will be used for detecting two events: the release instant of the master as well as the end of the second segment of the synchronization round.

The release instant occurs when aux_clock reaches the value Delta[clock_id]. This event forces the process to leave location Wait_delta and enter location Sync_second_half. It also causes a synchronization through the urgent channel rls_tm[clock_id] with the corresponding master SynM . After that, the automaton stays in location Sync_second_half until aux_clock reaches the value R1. Thus, while being in this location, the process works as an ideal timer. In the transition to the next location, the global variable num_clock is increased in one unit, whereas aux_clock is restarted. It also causes a synchronization through the urgent channel end_sync[clock_id] with the corresponding master SynM.

The temporal behavior of a master VC module is represented in Figure 9.16. This graph shows over a timeline all the events signaled by VC module as well as the global variables updated, which are used for interacting with the other processes of the model. The indexes have been eliminated for the sake of clarity, but remember that they provide a univocal correspondence between the processes VC module and SynM belonging to the same node.

Note that the synchronization round can be divided into three phases, which are labeled with

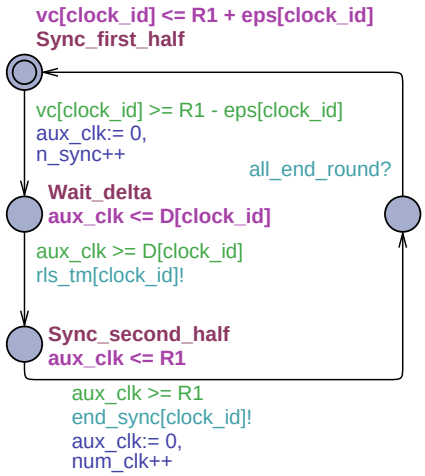


Figure 9.15: Final timed automaton of `VC module` (master)

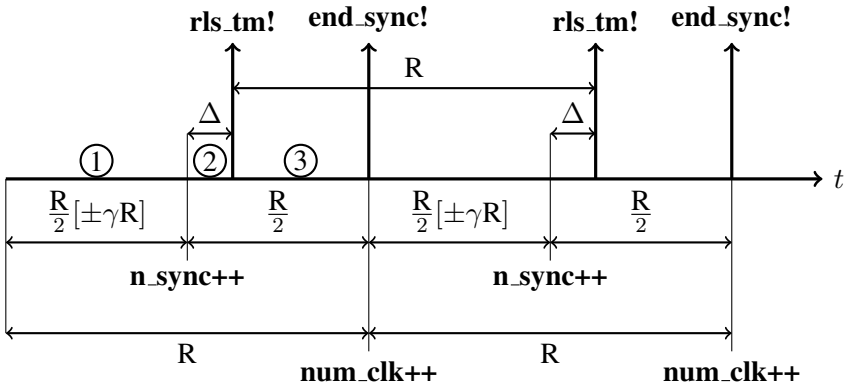


Figure 9.16: Temporal behavior of a master `VC module`

an encircled number. Although two synchronization rounds are actually depicted in the figure, only the phases of the first one are numbered. Phase 1 corresponds to the time spent in location `Sync_first_half` and in the location with no name, phase 2 corresponds to the spent in `Wait_delta` and, finally, phase 3 corresponds to the time spent in `Sync_second_half`.

The synchronization round starts in the transition from phase 1 to phase 2. In this transition, as already explained, the global variable `n_sync` is increased. This variable is monitored by the process `Observer` for measuring the precision, because the potential offset that the different processes `VC module` exhibit in the detection of this instant is a good approximation how much they may drift away. Note that phase 1 is measured by means of a perturbed timer and then its length may vary. In the graph we show the nominal duration of this phase.

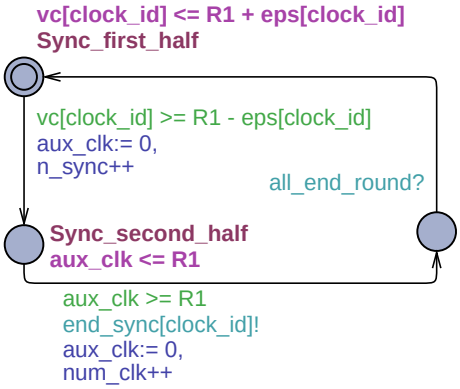


Figure 9.17: Final timed automaton of `VC module` (slave)

The transition to phase 3 happens Δ time units after the beginning of the round, and indicates that it is the time for the master to release its TM. The corresponding process `SynM` is notified of this event via the channel `rls_tm`. The release instants are theoretically periodical, and they occur with a nominal period of R time units, but since the length of phase 1 may vary, these instants may be advanced or delayed up to a certain amount: the offset with respect to the reference clock.

The end of phase 3 is not the end of the synchronization round, but the end of the time interval in which clock synchronization is possible. Masters are allowed to broadcast the TM only during one half of the synchronization round, and must remain idle during the other half. Using the notation of Figure 9.15, phase 1 corresponds to the idle half round, whereas phases 2 and 3 correspond to the active half round.

Note that the end of phase 3 is also notified to `SynM` via channel `end_sync`. At this instant, the global variable `num_clock` is increased. This variable is monitored by process `Round Ctrl` because there are some actions that must be performed once every node has reached the end of the active half round. These actions will be explained in Section 9.2.6.

The timed automaton of `VC module` for a slave, which is depicted in Figure 9.17, is a simplified version of the master's automaton. Notice that the slave does not have the location `Wait_delta` nor the synchronization through channel `rls_tm[clock_id]`. The rest of the automaton remains unmodified.

The temporal behavior resulting from this automaton is depicted in Figure 9.18. Note that, for the case of the slaves, phase 2 disappears. This is due to the fact that the process `SynM` of a slave does not have to broadcast the TM and therefore the signaling of the release instant is not required.

The similarities between the modeling pattern used for modeling the `VC module` of OCS-CAN

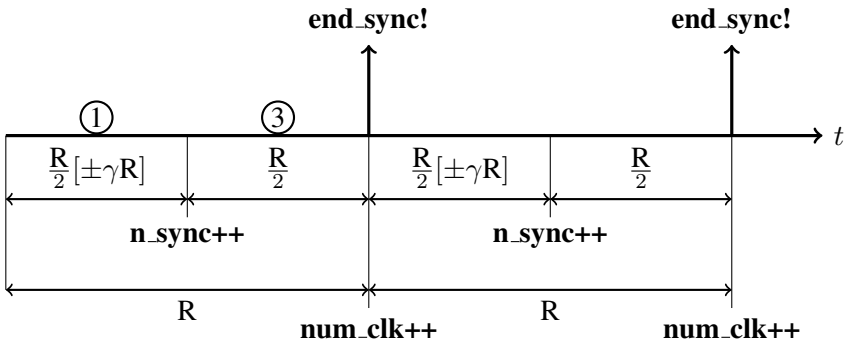


Figure 9.18: Temporal behavior of a slave `VC module`

and the modeling pattern discussed in Section 8.6 are very noticeable in Figure 9.18. For `VC module`, we also model the periodical execution of a task, and use two timers: one that behaves as a perturbed timer and another one that behaves as an ideal timer. Moreover, the offset exhibited by the perturbed timer will depend on the actions executed by the application, which in this case corresponds to process `SynM`.

Final model of process `SynM`

Once the whole modeling of `VC module` has been explained, it is possible to discuss the final model of process `SynM`. Remind that SynM can perform as a master or as a slave, and that the modeling of each kind of functionality is different. In Figure 9.19, we show the timed automaton of a process `SynM` master. Most of the features of this process were already described in Section 9.2; what is new in the automaton of Figure 9.19 concerns the modeling of the virtual clocks and thus the interaction with process `VC module`.

The first difference appears in the transition from `Idle1` to `Queue`. Note that this transition is now synchronized via the broadcast channel `rls_tm[my_id]`. As indicated in Section 9.2.6, this channel is signaled by the corresponding `VC module`.

The second difference is related to the the clock adjustment operation, which is caused by the reception of a high priority TM. Therefore, this change is visible both in the transition from `Idle1` to `Idle2` and in the transition from `Queue` to `Abort`. In such transitions, the clock adjustment operation is modeled with the algorithm shown in Algorithm 5.

However, before describing the actions performed in Algorithm 5, we need to clarify two important concepts used by `SynM`. This concepts are: the *reference clock* and the *time reference*. On the one hand, the reference clock is the identifier of the node to which a particular node is synchronized. It is

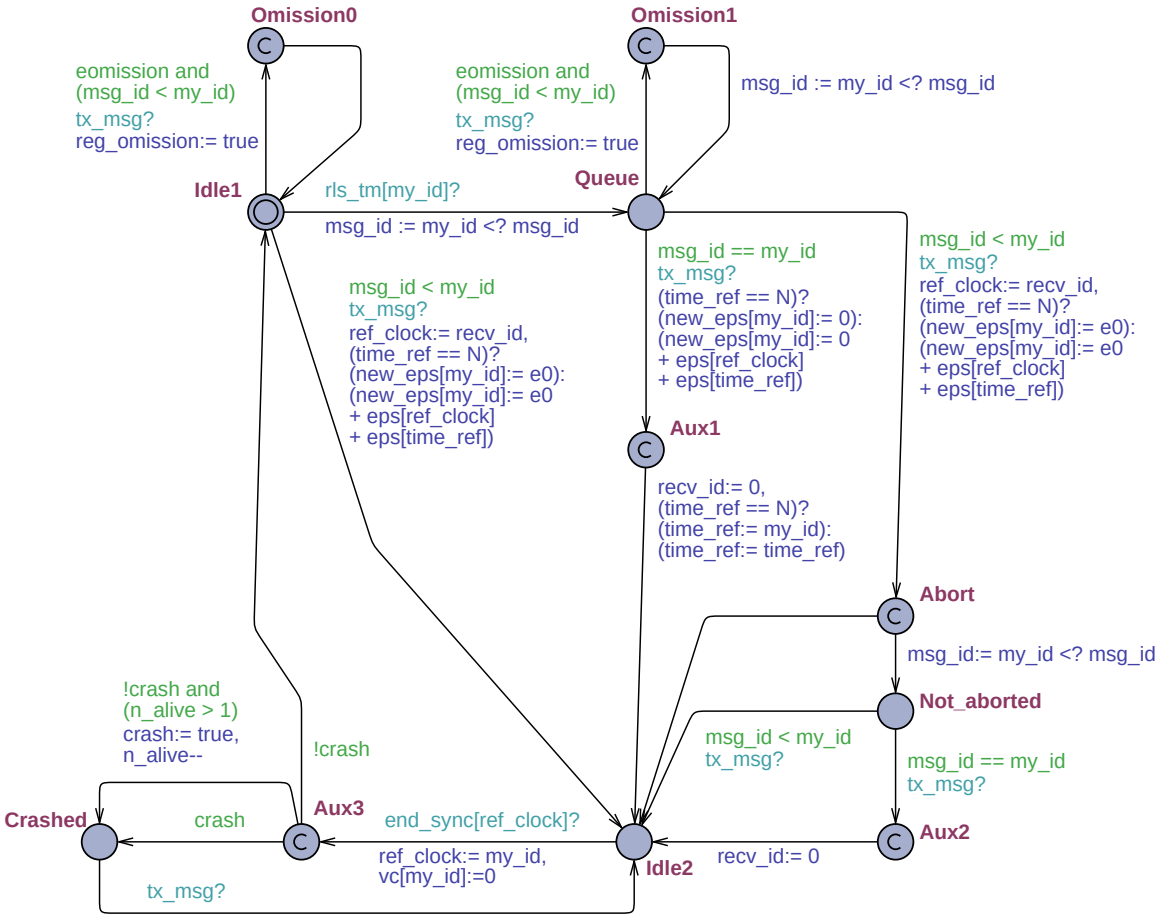


Figure 9.19: Final timed automaton of `SynM` (master)

Algorithm 5 Actions for modeling clock adjustment after receiving a high priority TM

ref_clock ← recv_id

if time_ref = N **then**

  new_eps[my_id] ← e0

else

  new_eps[my_id] ← e0 + eps[ref_clock] + eps[time_ref]

end if

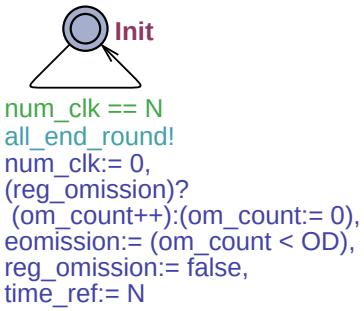


Figure 9.20: Final timed automaton of process `Round Ctrl`

modeled as a local variable, named `ref_clock`, and then may take a different value in each process `SynM`; it is equivalent to the reference clock used in the modeling patterns of Section 8.6.

On the other hand, the time reference is the identifier of the current active master of the system. The active master may change in every synchronization round, for instance as a result of a crash of the former active master, but it is consistently perceived by all processes `SynM` in the model. To ensure that consistency, it is specified as a global variable named `time_ref`.

By definition, the active master is the first master that successfully broadcasts its TM in a given round. Each node, either transmitter or receiver, can determine if a given TM is the first one transmitted/received in a round thanks to process `Round Ctrl`. This process, whose final version is shown in Figure 9.20, sets the global variable `time_ref` to its initial value (N) at the end of each synchronization round. Therefore, if a TM is transmitted/received and `time_ref` is equal to N then the identifier of the message indicates the active master of the round just beginning.

Thus, Algorithm 5, which is activated by the reception of a higher priority TM, starts by converting the transmitter of the just received TM into the reference clock of the node for the next round. After that, the master checks whether the transmitter of the TM is going to be the active master for the next round or not (condition `time_ref = N`). If it is not, it calculates the new offset (kept in variable `new_eps[my_id]`) but taking into account the offset with respect to the current active master.

It is important to remark that the value of `eps[time_ref]` is greater than 0 only if the active master has changed from the previous synchronization round. Therefore, as long as the active master does not change, this modeling pattern is equivalent to the one discussed in Section 8.6. Or, in other words, the modeling pattern finally adopted for OCS-CAN is actually an extension of the one discussed in Section 8.6, since it allows the abrupt change of the global time reference whereas the one of Section 8.6 assumed that Node 0 was always the active master.

The third difference is related to the successful transmission of the TM, happening in the transitions

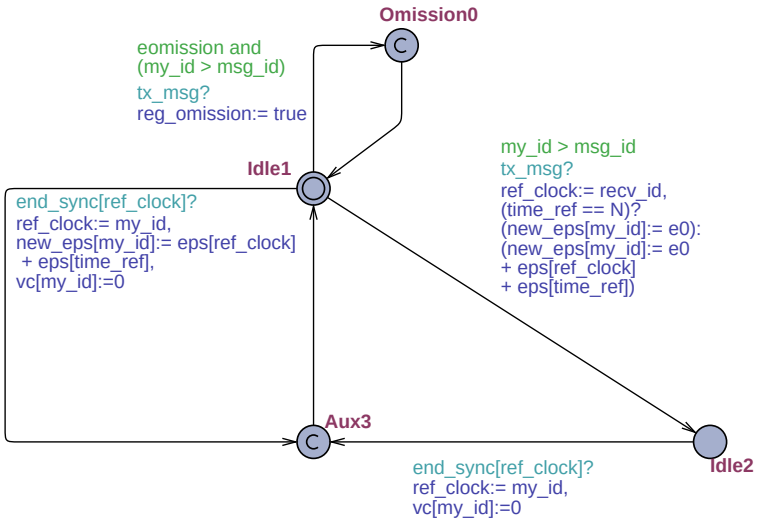


Figure 9.21: Final automaton of `SynM` (slave)

from `Queue` to `Aux1`. In this transition, `SynM` executes the actions shown in Algorithm 6 for calculating the new offset. Moreover, in the following transition (from `Aux1` to `Idle2`), process `SynM` checks again if it is the active master and updates variable `time_ref` accordingly.

Algorithm 6 Actions for modeling clock adjustment after successfully broadcasting a TM

if time_ref = N **then**

new_eps[my_id] ← 0

else

new_eps[my_id] ← eps[ref_clock] + eps[time_ref]

end if

The last difference concerns the detection of the end of the synchronization period, performed in the transition from `Idle2` to `Aux3`. In the final model of OCS-CAN, this event is signaled in principle by each `VC module` through its corresponding broadcast channel `end_sync[my_id]`. However, each `SynM` is synchronizing via the channel `end_sync` of its reference clock (kept in variable `ref_clock`). This is a direct application of the idea of *multiple clock pointers* defined in Section 8.6.

Figure 9.21 shows the timed automaton of a process `SynM` acting as a slave. This process is a simplification of the one defined for the master `SynM`. Given that the slaves do not broadcast any TM, the locations related to such broadcast (`Queue`, `Abort`, etc.) have been eliminated.

Apart from that, an additional transition (from `Idle1` to `Aux3`) has been included in the automaton. This transition is synchronized with channel `end_sync[ref_clock]` either, and serves the purpose of preventing the slave from getting stuck in location `Idle1` if no TM is received.

9.3 Verification procedure and results

This section constitutes the third and last part of the description of the model checking of OCS-CAN, in which we address the formal verification of the precision property. This property, as mentioned in Section 9.1.2, states that for any pair of non-faulty virtual clocks, the absolute value of their difference is never greater than a given value Π (the precision).

In Chapter 8 we provided a mechanism for measuring the offset between the computer clocks of a distributed system. This mechanism relied on an additional process, the so-called *Observer*, that measured the delay exhibited by the nodes (with respect to each other) when executing a periodical task. For the formal verification of OCS-CAN we will use the same mechanism; we will define an external process, called `Observer` as well, which will supervise the processes `VC module` in order to measure the separation exhibited by them when notifying a specific periodical instant: the start of the synchronization round.

After describing process `Observer`, we will discuss the scenarios that have been considered for formal verification. After that, the results that have been obtained for these scenarios will be discussed.

9.3.1 The process `Observer`

The process that we use for model checking the precision property in OCS-CAN is a variation of the process `Observer` discussed in Section 8.6.2. The timed automaton of the `Observer` of OCS-CAN is depicted in Figure 9.22. The main difference is that we have eliminated from the automaton the management of the inconsistent synchronization rounds, as they are already managed by process `Round Ctrl`.

The automaton has two locations: `Initial` and `Reached`. Location `Initial` is left as soon as one of the nodes reaches the beginning of the synchronization round. This happens whenever a process `VC module` leaves location `Sync_first_half`. As discussed in Section 9.2.6, in this transition the value of the global variable `n_sync` is increased. Due to this, process `Observer` defines a guard over variable `n_sync` for detecting this event. Furthermore, and in order to implement the aforementioned "as-soon-as" semantics, process `Observer` uses an urgent channel `a` for

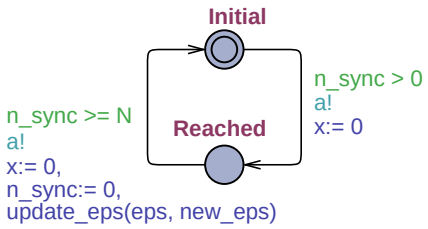


Figure 9.22: Final timed automaton of process `Observer`

synchronizing with a dummy automaton. This is equivalent to what was explained in Section 8.3.3.

Location `Reached` is left once all processes `VC module` have left location `Sync_first_half`, i.e. when `n_sync`= N. Thus, the maximum time that process `Observer` may stay in location `Reached`, as measured by the local clock `x`, corresponds to the maximum offset between the virtual clocks.

For determining the maximum value reachable by clock `x`, we ask the UPPAAL verifier to model check the following three properties.

```
A[] not deadlock
A[] Observer.Reached imply (Observer.x <= X)
E<> Observer.Reached and (Observer.x == X)
```

As indicated in Section 9.1.2, these properties have to be model checked iteratively for different values of X. However, given that the offset increases in steps of size e0, there are many intermediate values that can be neglected.

The maximum value of X that satisfies the three properties constitutes the upper bound of the offset and thus the precision of the system, for the considered parameters and fault assumptions.

9.3.2 Considered scenarios for formal verification

The formal model of OCS-CAN allows modification of the following parameters:

- Number of masters. This is configured with the global integer variable `N`.
- Resynchronization period (R). More specifically, it is configured with the global integer variable `R1`, which takes the value $\frac{R}{2}$.
- Release time of each master. This is set in the global array of integers `Delta[i]`.

- Residual error after clock correction (γ_0). This parameter gives a measure of the stability of the local oscillator that a specific virtual clock uses, and it also takes into account the error accumulated in the arithmetical operations of the clock adjustment procedure. This parameter is implicitly configured in the definition of the constant `e0`, since `e0`$= \gamma_0$R. In general, we assume the same value of γ_0 for each virtual clock, but the model allows these values to differ, if required.

- Network load. The variable `WCRT` in `Chan_ctrl` is used for specifying the worst-case response time of the TM. As already indicated, this parameter includes not only the delay caused by higher priority traffic on the bus, but also the delay caused by channel errors that lead to frame retransmissions.

- Master faults. The initial value of the global variable `n_alive` can be used for setting the maximum number of masters that may crash. For instance, if `n_alive` is initialized to 1, no master crash is allowed, whereas if `n_alive` is initialized to N then all but one masters may fail. No assumption is made on the order in which the masters crash.

- Data consistency. Inconsistent message omissions can be enabled/disabled in our model. Moreover, when they are enabled, it is possible to bound the maximum number of consecutive synchronization rounds that can suffer from inconsistencies. This is set with the constant `OD`. No assumption is made on the *spatial* distribution of the inconsistent omissions so that every possible combination of message inconsistency is checked.

Thanks to our parameterizable model of OCS-CAN, it is possible to generate and model check scenarios that correspond to the following situations:

1. Fault-free scenario
2. Only channel faults scenario, assuming *data consistency* and without assuming *data consistency*
3. Only master faults scenario
4. Master faults and channel faults scenario, assuming *data consistency* and without assuming *data consistency*

9.3.3 Results obtained and discussion

The first way of checking the correctness of a UPPAAL model is to use the simulator provided by the tool. However, it is also useful to model check certain properties that should hold during the

verification. This provides further guarantees that the model of the system is correctly built.

In our case, we model checked the following properties

(1) The maximum omission degree is never exceeded.

```
A[] o_count <= OD
```

(2) The maximum number of allowed master crashes is never exceeded.

```
A[] n_alive >= 1
```

(3) There is always at least one non-faulty master in the system (for N= 4 masters).

```
A[] not (Master0.crash and Master1.crash and
         Master2.crash and Master3.crash)
```

Concerning the precision guaranteed by the clock synchronization service, Table 9.1 shows the precision that was verified in different scenarios. These results were obtained with the following parameters: N= 4 masters, R= 1s, $\gamma_0 = 10^-6$ for each node, Δ_0= 0, Δ_1= 1 ms, Δ_2= 2 ms, Δ_3= 2 ms. Regarding the network load, it was assumed that no other messages where sent on the bus, so WCRT= 1.04 ms was used in those scenarios without channel faults whereas WCRT= 6 ms was used in those scenarios with channel faults. However, the results will show that the network load is not very relevant for the clock precision, as expected.

The first cell in Table 9.1 shows the precision guaranteed in the fault-free scenario. This precision equals to 2 μs. The first row of Table 9.1 corresponds to the scenarios in which only master's faults were assumed. Note that the number of faulty master does not affect significantly the precision guaranteed. Unfortunately, the limited temporal granularity of our model makes it impossible for us to measure the difference between having one or more faulty masters. Nevertheless, in some simplified verifications we have reported that this difference is in the order of 10 ns.

The first column of Table 9.1 corresponds to the scenarios in which only channel's faults were assumed. OD= 0 indicates that channel faults can occur, but that they may not cause any inconsistent omission; what is a common assumption in other clock synchronization protocols for CAN. The rest of cells in Table 9.1 correspond to the scenarios where a combination of node's and channel's faults is assumed. In particular, the right bottom cell corresponds to the most severe fault scenario.

Table 9.2 shows some of the results obtained when the resynchronization period is reduced to 0.5 s. These results prove the intuition that the negative effect of the inconsistencies may be reduced by synchronizing more frequently.

The results obtained show that certain failures have greater impact on the precision. Particularly, it is seen that inconsistent message omissions affect more negatively than master crashes.

Table 9.1: Fault assumptions and precision guaranteed (in μs) with R = 1 sec

# Channel faults	# Faulty masters			
	0	1	2	3
No faults	2	2	2	2
OD = 0	2	2	2	2
OD = 1	6	6	6	6
OD = 2	10	12	12	12
OD = 3	14	16	16	16

Table 9.2: Fault assumptions and precision guaranteed (in μs) with R = 0.5 sec

# Channel faults	# Faulty masters			
	0	1	2	3
OD = 0	1	1	1	1
OD = 1	3	3	3	3

Chapter 10

Conclusions and future work

This chapter summarizes the work presented in this dissertation and reviews the main contributions. It also presents the publications that have resulted from the presented work and provides some insight for further research.

10.1 Thesis validation and contributions

The motivation of our research was to demonstrate the thesis that: *"it is possible to design a clock synchronization service that fulfills the requirements for implementing dependable applications over CAN. The suitability of the clock synchronization service will be measured in terms of three attributes: high precision, fault tolerance and cost effectiveness"*. The work carried out for validating this thesis can be grouped in three main contributions:

1. Study of the state of the art concerning clock synchronization for dependable CAN.
2. Design and prototyping of OCS-CAN.
3. Formal assessment of OCS-CAN.

However, this dissertation also generated a fourth and unexpected contribution: the thorough study on the techniques for specifying distributed systems with computer clocks by means of timed automata. Although said study was not among our original goals, it has turned into an important contribution of our research, mainly because the modeling of computer clocks with timed automata is a problem with much more generality than just the formal verification of OCS-CAN.

These four contributions will be discussed in the next subsections.

10.1.1 Study of the state of the art concerning clock synchronization for dependable CAN

This study was performed in Chapter 5. This chapter started with a review of the solutions that have been proposed in the literature for solving the main limitations of CAN with respect to dependability. But we paid special attention to the relationship between said techniques and the provision of a clock synchronization service.

We observed that most of the techniques suggested for improving dependability in CAN —and more specifically, those aimed at reducing network jitter, improving error containment and supporting fault tolerance— assume the existence of a reliable clock service of high precision. Based on this observation, we claim that 1) clock synchronization plays a fundamental role in making the CAN fieldbus a suitable technology for dependable systems, and that 2) a clock synchronization service can be considered suitable for dependability only if it is fault-tolerant and achieves a precision in the order of a few μs, at a reasonable cost.

After determining the requirements of the clock synchronization service, in Chapter 5 we also studied the solutions currently available for implementing clock synchronization over CAN. We showed that none of these solutions exhibit the desired properties, particularly in what concerns fault tolerance. Although many authors have faced the problem of achieving high precision with satisfactory results, the provision of fault tolerance has not been properly addressed. In particular, we reported that the formal verification of these solutions has not been performed, and they have been evaluated only by means of simulation and testing.

Furthermore, we also reported that most of the solutions for clock synchronization cannot be considered independent with respect to the rest of the system. For instance, they may make strong assumptions about the communication pattern (some only work for event-triggered systems, whereas some others only work for time-triggered systems) or the hardware requirements (such as assuming the use of a certain CAN controller).

In our opinion, both the lack of independence and the lack of formal verification constitute severe impairments for adopting any of the available clock synchronization solutions as a component of a dependable system. As a conclusion, we claim that there is still room for a novel solution for clock synchronization over CAN, which should be independent of the system and must be formally assessed in order to ensure that the desired properties are guaranteed.

10.1.2 Design and prototyping of OCS-CAN

In Chapter 6 we proposed the architecture of OCS-CAN, a novel solution for clock synchronization over CAN. This solution pursues the achievement of four attributes that are important from the point of view of dependability: high precision, low overhead, fault tolerance and cost efficiency.

The architecture of OCS-CAN is made up of a set of independent hardware devices, the clock units, which are attached to the nodes. The great advantage of this approach is that the clock synchronization service is implemented as an *orthogonal subsystem*, which does not require the replacement of any existent component. Furthermore, OCS-CAN is compatible with any application since it makes no assumption about the system to which it is attached. Orthogonality not only helps to reduce the overall cost of the solution, but it is also very positive from a dependability perspective. The provision of clock synchronization by means of an orthogonal subsystem implies that the results of the formal assessment hold regardless of the system to which OCS-CAN is attached.

OCS-CAN is implemented in hardware because that reduces the system latencies, particularly during the transmission and reception of messages, and then improves the precision of the clock synchronization algorithm. The modified CAN controller that each clock unit incorporates adopts the idea given by Turski [Tur94] of signaling the SOF bit of each received frame. Additionally, and inspired by TTCAN Level 2 [FMD+00], we define a hardware-implemented low level mechanism for timestamping the messages sent and received by each clock unit.

OCS-CAN indirectly lowers its own cost by reducing the communication and computation overheads. In our proposal this is achieved by implementing an asymmetric (i.e. master/slave) scheme of synchronization. Moreover, thanks to the hardware mechanism for timestamping, clock synchronization can be achieved in the best case with only one message per round.

In order to provide tolerance to faults of the master, some of the clock units are configured to work as backup masters that may replace the active master if it fails. The algorithm for managing master redundancy takes advantage of two properties of CAN, the arbitration and the bounded response time, for reducing the communication overhead.

The fault-tolerant algorithm of OCS-CAN assumes that the clock units exhibit *crash* failure semantics. It is possible to ensure this property by means of internal duplication and comparison. However, the specific way to implement said duplication and comparison scheme has not been addressed in this dissertation.

The development of the first prototype of OCS-CAN was indeed discussed in Chapter 6. The aim of our prototype was to prove that OCS-CAN can be implemented with low-cost hardware, such as a medium range FPGA. The precision measured in the tests was impaired by the fact of using physical clocks of low frequency, but the results were acceptable. Moreover, as it was discussed in

our previous study, we know that the achievement of high precision is not the main difficulty of this research; many authors have reported good enough results in fault-free conditions. For this reason, we preferred to focus on the evaluation of the fault tolerance techniques, which has not been addressed in the literature and is therefore a more innovative contribution.

10.1.3 Formal assessment of OCS-CAN

In this dissertation we have assessed in two different ways the precision guaranteed by OCS-CAN in the presence of faults. Whereas in Chapter 7 the assessment is performed analytically, in Chapter 9 it is performed by means of model checking with UPPAAL.

The analysis of Chapter 7 has been addressed by approximating the virtual clocks of OCS-CAN as piecewise linear functions with points of discontinuity that coincide with the synchronization instants. The aim of our analysis was to find the equations that relate the guaranteed precision with the relevant parameters of the clock synchronization algorithm, such as the synchronization period, the number of backup masters, etc., as well as with the considered fault assumptions.

This aim has been successfully fulfilled, at least for the most likely fault scenarios. However, the presented analysis shows an inherent difficulty: it forces us to figure out the worst case scenario by reasoning about the system behavior. Although finding such a scenario can be straightforward in many situations, it can be unfeasibel under some fault conditions, for instance when a complex combination of channel and node faults may occur. Due to this, we complemented our analysis with the formal verification of OCS-CAN by means of model checking.

Model checking is a technique that automatically generates and examines all possible traces of a system's model. Therefore, model checking frees the system modeler from finding the worst case scenario. Given that model checking evaluates *all* possible scenarios, it is sure that the worst case is going to be found and evaluated. Nevertheless, it requires the user to build a correct model of the system, in which all the relevant behaviors are specified.

For the case of OCS-CAN, this implies building a proper model of the CAN network as well as a proper model of the clock units. These models must also include the potential faults of the system's components, since our aim is mainly to evaluate the fault tolerance mechanisms.

Among the available tools for model checking, we have used the UPPAAL model checker, which is a tool based on the theory of *timed automata*, and incorporates a quantitative notion of time in the form of clock variables. Although the use of timed automata represented a challenge for the modeling of drifting virtual clocks and clock synchronization actions, it was possible to overcome the limitations of this formalism and specify the computer clocks appropriately. For that, we developed a new modeling pattern for timed automata, which we have called *clock pointers*.

The results of the model checking of OCS-CAN show that the algorithm for managing master redundancy ensures a certain precision in faulty conditions. It also shows the *graceful degradation* of the precision in the presence of faults, given that the precision among the virtual clocks worsens gradually as the fault scenarios become more severe.

It is worth noting that our approach for evaluating OCS-CAN is quite different from the approach followed in order to evaluate other fault-tolerant clock synchronization algorithms for CAN, such as [RGR98, HMF+00, LA03] and [APSP07]. These solutions were evaluated only by means of simulation and testing.

In contrast, the main strength of our evaluation of OCS-CAN lays on the fact that it allows a quantitative assessment of the effect that faults may have on the system precision. This allows the system designer to easily configure the system in a way such that the requirements on the precision are fulfilled. Moreover, our analysis helps to understand in which conditions the desired precision may be lost and, thus, helps to identify potential risks.

In the literature about fault-tolerant clock synchronization in distributed systems, it is possible to find several analyses that study the precision of certain algorithms under different fault assumptions; for instance in [ST85, MS85, Sch87] and [FC95], among many others. Nevertheless, the results of these papers are not applicable to OCS-CAN, since they only address clock synchronization algorithms that provide fault tolerance by means of massive message exchanges. They focus on proving the correctness of their *converge functions*, but their results are not extensible to the master/slave schemes since these schemes use only one message per round and do not apply any converge function. The same can be said about the formal verification carried out in [PSvH99], which assumes that in every synchronization round several synchronization messages are sent and that, therefore, every processor counts on several timestamps per round to adjust its clock.

The results obtained in our assessment show that certain failures have greater impact on the precision. Particularly, we observed that inconsistent message omissions affect more negatively than master crashes. It is curious that even though master crashes are usually addressed by current solutions for clock synchronization, very little attention is paid to the fact that message inconsistencies may occur [FMD+00]. It may be argued that message inconsistencies are very unlikely in CAN networks, but certain authors claim that the probability is such that it should be taken into account when designing fault-tolerant systems for dependable applications [RVA+98, PMJ00]. Moreover, it has been reported that the probability of message inconsistencies in TTCAN increases dramatically when compared to the so-called natural CAN [RNP03a].

To the author's best knowledge, this dissertation is the first one to explicitly address the analysis of fault-tolerant master/slave clock synchronization in the presence of inconsistent message omissions. We believe that the techniques we adopted for the analysis of OCS-CAN may be interesting for

the analysis of other fault-tolerant master/slave clock synchronization systems. It seems that the master/slave scheme may be one of the preferred solutions for implementing clock synchronization in many low-cost distributed embedded systems. For instance, the fault-tolerant algorithm proposed in the definition of the IEEE1588 standard [IEE02] for clock synchronization for control systems (also known as PTP-*Precise Time Protocol*) relies on master/slave. Therefore, it is susceptible of being analyzed with the mathematical framework developed in our research.

10.1.4 Modeling patterns for distributed systems with computer clocks

As discussed in Chapter 8, the model checking of OCS-CAN forced us to address one of the most important limitations of the theory of timed automata. Timed automata only allow, at least theoretically, specification of clocks that evolve at the rate of real time, but for the formal verification of OCS-CAN we had to specify computer clocks that, not only evolve at a rate that differs from real time, but also may change their values and rates abruptly as a consequence of the synchronization actions. In order to overcome this limitation, we first investigated the applicability of some modeling techniques already available in the literature on timed automata, such as the *perturbed timed automata*. But finally we had to develop a novel modeling technique, which we have called *clock pointers*. This technique particularly allowed us to specify distributed systems in which the nodes may follow different time references during transient time intervals.

Given that the problem of specifying computer clocks with timed automata is not exclusive of OCS-CAN, but it can be found when modeling many kinds of distributed systems, we devoted Chapter 8 to describing these modeling techniques in an exhaustive and systematic way. The aim of this description was to define a series of modeling patterns that can be adopted by other researchers, and to explain and justify for what kind of systems they are useful. For describing each modeling pattern, we made use of graphs that showed the evolution of the timed automata clocks (and thus of the models) in a very intelligible way. Additionally, formal proofs in the form of temporal logic properties were provided for demonstrating the correctness of the patterns suggested.

In our opinion, our comprehensive description of these modeling patterns is a significant contribution of this research. First, because it sheds some light on an important topic —the realistic specification of the temporal behavior of the nodes of a distributed system— which has to a certain extent remained obscure, at least for the average user of model checking. Second, because it introduces a new modeling pattern, the clock pointers, which extends a bit further the applicability of the timed automata formalism.

10.2 Publication of results

This section gathers the different publications we have authored and that are related to this dissertation. Some of these publications resulted directly from the work herein presented, whereas some others have a more indirect relationship.

10.2.1 Preliminary publications

These publications are the result of some research we undertook in parallel to the work described in this dissertation. They were fundamental for understanding the role that clock synchronization plays for the achievement of dependability in CAN, as well as for understanding the relevance of providing suitable fault tolerance mechanisms.

Peer-reviewed papers published in international scientific journals:

- I. Broster, A. Burns, and G. Rodríguez-Navas. Timing analysis of real-time communication under electromagnetic interference. *Real-Time Systems*, 30(1-2):55–81, 2005.
- J. Ferreira, L. Almeida, J.A. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza. Combining operational flexibility and dependability in FTT-CAN. *Industrial Informatics, IEEE Transactions on*, 2(2):95–102, 2006.

Peer-reviewed papers published in international conferences:

- I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. *Proceedings of the 23rd Real-time Systems Symposium*, 2002.
- G. Rodriguez-Navas and J. Proenza. Analyzing Atomic Broadcast in TTCAN Networks. In *Proc. of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FET'03)*, 2003.
- G. Rodriguez-Navas, M. Barranco, and J. Proenza. Harmonizing Dependability and Real Time in CAN Networks. In *2nd Euromicro International Workshop in Real-Time LANS in the Internet Age*, pages 47–50, 2003.
- G. Rodríguez-Navas, M. Barranco, J. Proenza, and I. Broster. COTS-based hardware support to timeliness in CAN networks. In *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003)*, Lisbon, Portugal, Sept 2003. IEEE.

10.2.2 Publications of results presented in this dissertation

The following publications spread the main results and contributions presented in this dissertation.

Peer-reviewed papers published in international scientific journals:

- G. Rodriguez-Navas, S. Roca, and J. Proenza. Orthogonal, Fault-Tolerant and High-Precision Clock Synchronization for the Controller Area Network. *IEEE Transactions on Industrial Informatics*, 4(2):92–101, May 2008.

Peer-reviewed papers published in international conferences:

- G. Rodriguez-Navas and J. Proenza. An orthogonal and fault-tolerant subsystem for high-precision clock synchronization in CAN networks. In *Proceedings of the 10th WSEAS International Conference on Signal Processing, Robotics and Automation (ISPRA), Chiclana, Cádiz (Spain)*, 2002.

- G. Rodríguez-Navas, J. Bosch, and J. Proenza. Hardware Design of a High-precision and Fault-tolerant Clock Subsystem for CAN Networks. *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2003), Aveiro, Portugal*, 2003.

- G. Rodríguez-Navas and J. Proenza. Clock Synchronization in CAN Distributed Embedded Systems. *Proc. of the 3rd. International Workshop on Real-Time Networks, Catania, Italy*, 2004.

- G. Rodríguez-Navas, J. Proenza, and H. Hansson. Using UPPAAL to Model and Verify a Clock Synchronization Protocol for the Controller Area Network. *Proc. of the 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy*, 2005.

- G. Rodriguez-Navas, J. Proenza, and H. Hansson. An UPPAAL Model for Formal Verification of Master/Slave Clock Synchronization over the Controller Area Network. In *Proc. of the 6th IEEE International Workshop on Factory Communication Systems, Torino, Italy*, 2006.

- G. Rodriguez-Navas, J. Proenza, and H. Hansson. Modeling and verification of master/slave clock synchronization using hybrid automata and model-checking. In *Proceedings of the 9th International Conference on Formal Engineering Methods (ICFEM 2007), LNCS 4789*, pages 307–326, 2007.

- G. Rodriguez-Navas, J. Proenza, and H. Hansson. Analytical assessment of the precision degradation caused by faults in a fault-tolerant master/slave clock synchronization service for CAN.

In *Proceedings of the 23th IEEE International Symposium on Reliable Distributed Embedded systems (SRDS'08), Napoli (Italy)*, 2008.

Book chapters:

- J. Pimentel, J. Proenza, L. Almeida, G. Rodriguez-Navas, M. Barranco, and J. Ferreira. *Dependable Automotive CAN Networks (chapter in Automotive Embedded Systems Handbook)*. CRC Press, 2009.
- G. Rodriguez-Navas, J. Proenza, Hans Hansson, and Paul Pettersson. *Using Timed Automata for Modeling the Clocks of Distributed Embedded Systems (chapter in Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation)*. IGI Global, 2010.

Technical reports:

- G. Rodríguez-Navas and J. Proenza. On the design of a clock service for CAN networks. Technical Report A-1-2003, Universitat de les Illes Balears, 2003.
- G. Rodríguez-Navas, J. Proenza, and H. Hansson. An UPPAAL Model for Formal Verification of Clock Synchronization over CAN. Technical Report A-2-2005, Universitat de les Illes Balears, 2005.

10.3 Future work

This section discusses some ideas for future work, which might extend the work presented in this dissertation. More specifically, we can point out two clear directions for further research. The first direction consists in extending and complementing the design and evaluation of OCS-CAN, whereas the second direction consists in applying the techniques that have been developed for OCS-CAN, but for the implementation and evaluation of other kinds of distributed systems. Both possible directions are briefly discussed next.

10.3.1 Possible extensions of the work on OCS-CAN

With regard to the architecture of OCS-CAN, there is one aspect that has not been covered in this dissertation: the design of the mechanism for restricting the failure semantics of the clock units to

crash failure semantics. As indicated in Chapter 6, this assumption can be substantiated by means of an internal mechanism for duplication and comparison, for instance as it is described in [PPMJ99]. Other options would involve the adoption of external elements, such as *bus guardians* [FAF+06] or a central hub with error-detection capabilities [BPRNA06], but the suitability of such techniques must still be evaluated.

Concerning the model checking of OCS-CAN, we are investigating the way to generalize the results obtained with UPPAAL to an arbitrary number of masters. Apparently, the Omission Degree is the limiting factor with regard to the precision, because inconsistent message omissions are the main cause of precision degradation. Due to this, we hypothesize that, as long as a minimum number of masters are present, the precision achievable neither improves or worsens by defining more masters.

Another interesting way to extend our work on OCS-CAN is by quantifying its reliability. The formal assessment presented in this dissertation allows us to quantify the precision degradation caused by faults, and then to identify the most dangerous scenarios. A reliability analysis could complement this study by providing the probabilities of those scenarios. The combination of these two techniques then would yield a probability distribution of the precision, which would be very useful in order to integrate OCS-CAN as a component of some dependable CAN-based systems.

For quantifying the reliability analysis of OCS-CAN, we are considering the application of the formalism known as *Stochastic Activity Network* (SAN), which is a extension to *Stochastic Petri Nets* [CFJ+91]. Our research group has had some successful experiences with this formalism, as it has been recently used for evaluating two novel star topologies over CAN [Bar10], and we expect to proceed similarly with OCS-CAN. Another appealing approach for reliability evaluation would be to investigate the possibility of using probabilistic model checking as a means to perform both formal verification and reliability analysis simultaneously.

10.3.2 Potential applications of the developed techniques

The most direct application of the work presented in this dissertation is obviously the integration of OCS-CAN with other solutions for improving the dependability properties of CAN. In this sense, we are currently participating in a project called *CAN-based Infrastructure for Dependable Systems* (CANbids) that aims precisely at proposing a complete architecture for supporting dependable application over CAN. One of the first goals of this project is to investigate the possible integration of OCS-CAN with two other proposals for improving the dependability of CAN: FTT-CAN [APF02] and CANcentrate [BPRNA06]. These three solutions have a common characteristic, they all rely on a central element with a privileged view of the system (the OCS-CAN master, the FTT-CAN master and the CANcentrate hub) and define redundancy for tolerating faults of said central element. We

plan to investigate the possibility of integrating all these functionalities into a single node.

With respect to the formal assessment of OCS-CAN, it is worth recalling that some of the techniques developed in this dissertation have broader applicability than just the evaluation of OCS-CAN. Particularly, we would like to highlight three aspects of the presented work that in our opinion merit further investigation:

- The analytical framework presented in Chapter 7 could be applied for the evaluation of other master/slave clock synchronization algorithms, especially if they may suffer from message inconsistencies.
- The techniques described in Chapter 9 for modeling the CAN network with UPPAAL, which included the modeling of message broadcast, arbitration, message abortion, message inconsistencies, etc., could be applied by other researchers for model checking their own CAN-based systems.
- As already indicated in Chapter 8, the patterns we have discussed for modeling computers clocks could be applied for modeling many types of distributed systems. But they could also be applicable for the formal verification of other kinds of *hybrid systems*, such as control systems working with physical systems that exhibits some uncertainty.

Bibliography

[ABG+08] S. Akshay, Benedikt Bollig, Paul Gastin, Madhavan Mukund, and K. Narayan Kumar. Distributed timed automata with independently evolving clocks. In *CONCUR '08: Proceedings of the 19th international conference on Concurrency Theory*, pages 82–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, 1993.

[ADMB00] Eugene Asarin, Thao Dang, Oded Maler, and Olivier Bournez. Approximate reachability analysis of piecewise-linear dynamical systems. In *HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 20–31, London, UK, 2000. Springer-Verlag.

[Alt] Altera Corp. website.

[AP97] E. Anceaume and I. Puaut. A taxonomy of clock synchronization algorithms. Technical Report IRISA-PI - 97-1103, IRISA, 1997.

[APF02] L. Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics*, 49(6), 2002.

[APSP07] D. Ayavoo, M. J. Pont, M. Short, and S. Parker. Two novel shared-clock scheduling algorithms for use with 'Controller Area Network' and related protocols. *Microprocessors and Microsystems*, 31(5):326–334, 2007.

[ATM05] R. Alur, S. La Torre, and P. Madhusudan. Perturbed Timed Automata. In M. Morari and L. Thiele, editors, *8th International Workshop, Hybrid Systems: Computation and Control, HSCC 2005*, number 3414 in LNCS, pages 70–85. Springer–Verlag, March 2005.

[Bar10] M. Barranco. *Improving Error Containment and Reliability of Communication Subsystems Based on Controller Area Network CAN by Means of Adequte Star Topologies*. PhD thesis, Universitat de les Illes Balears, 2010.

[BB01] I. Broster and A. Burns. Timely use of the CAN protocol in critical hard real-time systems with faults. *Proceedings of the 13th Euromicro Conference on Real-time Systems, Delft, The Netherlands*, 2001.

[BB03] I. Broster and A. Burns. An analysable bus-guardian for event-triggered communication. *Proc. of the 24th Real-Time Systems Symposium, Cancun, Mexico*, 2003.

[BBRN02] I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. *Proceedings of the 23rd Real-time Systems Symposium*, 2002.

[BBRN05] I. Broster, A. Burns, and G. Rodríguez-Navas. Timing analysis of real-time communication under electromagnetic interference. *Real-Time Systems*, 30(1-2):55–81, 2005.

[BDH+06] Gerd Behrmann, Alexandre David, John Håkansson, Martijn Hendriks, Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal 4.0. In *Proceedings of the 3*rd *International Conference on Quantitative Evaluation of Systems*, 2006.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BFK+98] H. Bowman, G. Faconti, J-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip-synchronisation algorithm using UPPAAL - extended version. In *Third Internatinoal Workshop on Formal Methods for Industrial Crtical Systems, FMICS'98*, 1998.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BPRNA06] M. Barranco, J. Proenza, G. Rodriguez-Navas, and L. Almeida. An active star topology for improving fault confinement in CAN networks. *Industrial Informatics, IEEE Transactions on*, 2(2):78–85, 2006.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs, LNCS vol.131*, 1982.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[CFJ+91] Joseph A. Couvillion, Roberto Freire, Ron Johnson, W. Douglas Obal, M. Akber Qureshi, Manish Rai, William H. Sanders, and Janet E. Tvedt. Performability modeling with ultrasan. *IEEE Software*, 8(5):69–80, 1991.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 2001.

[CiA] Website of the CAN in Automation (CiA) group.

[DBBL07] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[DL07] Cătălin Dima and Ruggero Lanotte. Distributed time-asynchronous automata. In *ICTAC'07: Proceedings of the 4th international conference on Theoretical aspects of computing*, pages 185–200, Berlin, Heidelberg, 2007. Springer-Verlag.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95),* Pisa, Italy, pages 66–75, 1995.

[Ets01] K. Etschberger. *Controller Area Network*. IXXAT Press, Weingarten, 2001.

[FAF+06] J. Ferreira, L. Almeida, J.A. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza. Combining operational flexibility and dependability in FTT-CAN. *Industrial Informatics, IEEE Transactions on*, 2(2):95–102, 2006.

[FAM+03] J. Ferreira, L. Almeida, Ernesto Martins, P. Pedreiras, and J. Fonseca. Components to Enforce Fail-Silent Behavior in Dynamic Master-Slave Systems. *Proceedings of the* 5^{th} *IFAC Int. Symposium on Intelligent Components and Instruments for Control Applications*, 2003.

[FC95] Christof Fetzer and Flaviu Cristian. An optimal internal clock synchronization algorithm. In *Compass '95: 10th Annual Conference on Computer Assurance*, pages 187–196, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[Fer08] Sebastià Roca Ferrer. Implementació i test d'un sistema de sincronització de rellotge sobre CAN. Projecte Final de Carrera, Eng. Tècnica Industrial, especialitat Electrònica Industrial, UIB, 2008.

[FMD+00] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther, and Robert Bosch GmbH. Time Triggered Communication on CAN. *Proceedings of the 7th Int. CAN Conference, Amsterdam, The Netherlands*, 2000.

[GS94] M. Gergeleit and H. Streich. Implementing a Distributed High-resolution Real-Time Clock using the CAN-bus. *Proceedings of the 1st International CAN Conference, Mainz, Germany*, 1994.

[HKPV98] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[HMF+00] F. Hartwich, B. Müller, T. Führer, R. Hugel, and Robert Bosch GmbH. CAN network with Time Triggered Communication. *Proceedings of the 7th International CAN Conference, Amsterdam, The Netherlands*, 2000.

[IEE02] IEEE-1588. *Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Instrumentation and Measurement Society, 2002.

[ISO93] ISO. ISO11898. Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication, 1993.

[JBS07] Susmit Jha, Bryan A. Brady, and Sanjit A. Seshia. Symbolic reachability analysis of lazy linear hybrid automata. In *FORMATS'07: Proceedings of the 5th international conference on Formal modeling and analysis of timed systems*, pages 241–256, Berlin, Heidelberg, 2007. Springer-Verlag.

[Kat98] J.-P. Katoen. Concepts, algorithms, and tools for model checking. Lecture Notes of the course Mechanised Validation of Parallel Systems, 1998.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Press, 1997.

[LA03] D. Lee and G. Allan. Fault-tolerant Clock synchronisation with Microsecond-precision for CAN Networked Systems. *Proceedings of the 9th International CAN Conference, Munich, Germany*, 2003.

[LH06] G. Leen and D. Heffernan. Modeling and verification of a time-triggered networking protocol. *ICNICONSMCL*, 0:178, 2006.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[Mah01] N. P. Mahalik. *Fieldbus Technology*. Springer, 2001.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MS85] Stephen R. Mahaney and Fred B. Schneider. Inexact agreement: accuracy, precision, and graceful degradation. In *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 237–249, 1985.

[NNH05] T. Nolte, M. Nolin, and H. A. Hansson. Real-time server-based communication with CAN. *Industrial Informatics, IEEE Transactions on*, 1(3):192–201, 2005.

[PMJ00] J. Proenza and J. Miro-Julia. MajorCAN: A modification to the Controller Area Network to achieve Atomic Broadcast. *IEEE Int. Workshop on Group Communication and Computations. Taipei, Taiwan*, 2000.

[PPA+09] J. Pimentel, J. Proenza, L. Almeida, G. Rodriguez-Navas, M. Barranco, and J. Ferreira. *Dependable Automotive CAN Networks (chapter in Automotive Embedded Systems Handbook)*. CRC Press, 2009.

[PPMJ99] J. Proenza, J. Pons, and J. Miro-Julia. A low-cost fail-safe circuit for fault-tolerant control systems. In *Proc. of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 1999.

[Pro07] J. Proenza. *RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance*. PhD thesis, Universitat de les Illes Balears, 2007.

[PSvH99] H. Pfeifer, D. Schwier, and F. v. Henke. Formal verification for time triggered clock synchronization. *Proceedings of the 7th IFIP International Conference on Dependable Computing for Critical Applications, San Jose, CA*, 1999.

[Pur00] A. Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.

[PV03] L.M. Pinho and F. Vasques. Reliable Real-Time Communication in CAN Networks. *IEEE Transactions on Computers*, 52(12):1594–1607, 2003.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RGR98] L. Rodrigues, M. Guimaraes, and J. Rufino. Fault-tolerant Clock Synchronization in CAN. *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain*, 1998.

[RNBP03a] G. Rodriguez-Navas, M. Barranco, and J. Proenza. Harmonizing Dependability and Real Time in CAN Networks. In *2nd Euromicro International Workshop in Real-Time LANS in the Internet Age*, pages 47–50, 2003.

[RNBP03b] G. Rodríguez-Navas, J. Bosch, and J. Proenza. Hardware Design of a High-precision and Fault-tolerant Clock Subsystem for CAN Networks. *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2003), Aveiro, Portugal*, 2003.

[RNBPB03] G. Rodríguez-Navas, M. Barranco, J. Proenza, and I. Broster. COTS-based hardware support to timeliness in CAN networks. In *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003)*, Lisbon, Portugal, Sept 2003. IEEE.

[RNP02] G. Rodriguez-Navas and J. Proenza. An orthogonal and fault-tolerant subsystem for high-precision clock synchronization in CAN networks. In *Proceedings of the 10th WSEAS International Conference on Signal Processing, Robotics and Automation (ISPRA), Chiclana, Cádiz (Spain)*, 2002.

[RNP03a] G. Rodriguez-Navas and J. Proenza. Analyzing Atomic Broadcast in TTCAN Networks. In *Proc. of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FET'03)*, 2003.

[RNP03b] G. Rodríguez-Navas and J. Proenza. On the design of a clock service for CAN networks. Technical Report A-1-2003, Universitat de les Illes Balears, 2003.

[RNP04] G. Rodríguez-Navas and J. Proenza. Clock Synchronization in CAN Distributed Embedded Systems. *Proc. of the 3rd. International Workshop on Real-Time Networks, Catania, Italy*, 2004.

[RNPH05a] G. Rodríguez-Navas, J. Proenza, and H. Hansson. An UPPAAL Model for Formal Verification of Clock Synchronization over CAN. Technical Report A-2-2005, Universitat de les Illes Balears, 2005.

[RNPH05b] G. Rodríguez-Navas, J. Proenza, and H. Hansson. Using UPPAAL to Model and Verify a Clock Synchronization Protocol for the Controller Area Network. *Proc. of the 10th*

IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy, 2005.

[RNPH06] G. Rodriguez-Navas, J. Proenza, and H. Hansson. An UPPAAL Model for Formal Verification of Master/Slave Clock Synchronization over the Controller Area Network. In *Proc. of the 6th IEEE International Workshop on Factory Communication Systems, Torino, Italy*, 2006.

[RNPH07] G. Rodriguez-Navas, J. Proenza, and H. Hansson. Modeling and verification of master/slave clock synchronization using hybrid automata and model-checking. In *Proceedings of the 9th International Conference on Formal Engineering Methods (ICFEM 2007), LNCS 4789*, pages 307–326, 2007.

[RNPH08] G. Rodriguez-Navas, J. Proenza, and H. Hansson. Analytical assessment of the precision degradation caused by faults in a fault-tolerant master/slave clock synchronization service for CAN. In *Proceedings of the 23th IEEE International Symposium on Reliable Distributed Embedded systems (SRDS'08), Napoli (Italy)*, 2008.

[RNPHP10] G. Rodriguez-Navas, J. Proenza, Hans Hansson, and Paul Pettersson. *Using Timed Automata for Modeling the Clocks of Distributed Embedded Systems (chapter in Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation)*. IGI Global, 2010.

[RNRP08] G. Rodriguez-Navas, S. Roca, and J. Proenza. Orthogonal, Fault-Tolerant and High-Precision Clock Synchronization for the Controller Area Network. *IEEE Transactions on Industrial Informatics*, 4(2):92–101, May 2008.

[RVA+98] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. *Digest of papers, The 28th IEEE International Symposium on Fault-Tolerant Computing, Munich, Germany*, 1998.

[RVA99] J. Rufino, P. Veríssimo, and G. Arroz. A Columbus' egg idea for CAN media redundancy. *Digest of Papers, The 29th International Symposium on Fault-Tolerant Computing Systems*, 1999.

[SC90] Frank Schmuck and Flaviu Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 133–143, New York, NY, USA, 1990. ACM Press.

[Sch87] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report TR 87–859, Dept. of Computer Science, Upson Hall, Ithaca, NY 14853, 1987.

[SP07] M. Short and M.J. Pont. Fault-Tolerant Time-Triggered Communication Using CAN. *Industrial Informatics, IEEE Transactions on*, 3(2):131–142, 2007.

[ST85] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. In *Symposium on Principles of Distributed Computing*, pages 71–86, 1985.

[TBW95] K. Tindell, A. Burns, and A. J. Wellings. Calculating Controller Area Network (CAN) Message Response Time. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[Tör95] M. Törngren. A perspective to the Design of Distributed Real-time Control Applications based on CAN. *Proceedings of 2nd International CAN Conference, London-Heathrow, United Kingdom*, 1995.

[Tur94] K. Turski. A global time system for CAN. *Proceedings of the 1st International CAN Conference, Mainz, Germany*, 1994.

[Ver94] Paulo Veríssimo. Ordering and timeliness requirements of dependable real-time programs. *Real-Time Systems*, 7(2):105–128, 1994.

[VR01] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Springer, 2001.

[WDMR08] Martin Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. Robust safety of timed automata. *Form. Methods Syst. Des.*, 33(1-3):45–84, 2008.