TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
48/2020

Model-Based Testing of Real-Time Distributed Systems

DEEPAK PAL

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science
The dissertation was accepted for the defence of the degree of Doctor of Philosophy in Informatics on 22 October 2020

Supervisor: Professor Jüri Vain,
Department of Software Science,
Tallinn University of Technology,
Tallinn, Estonia

Opponents: Professor Johan Lilius,
Department of Information Technologies,
Åbo Akademi University,
Turku, Finland

Professor Artem Boyarchuk,
Department of Computer Systems and Networks,
Kharkiv Aerospace University,
Kharkiv, Ukraine

Defence of the thesis: 10 December 2020, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Deepak Pal _____
signature

ISSN 2585-6898 (publication)
ISBN 978-9949-83-635-2 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-636-9 (PDF)
Printed by Koopia Niini & Rauam

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
48/2020

Reaalaja hajussüsteemide mudelipõhine testimine

DEEPAK PAL

TAL TECH
KIRJASTUS

Contents

List of Publications

The present Ph.D. thesis is based on the following publications that are referred to in the text by Roman numbers.

I Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Automatic Distribution of Local Testers for Testing Distributed Systems. In: Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. (Ed.). Databases and Information Systems ix: Selected Papers from the Twelfth International Baltic Conference, DB&IS 2016 (297-310). Amsterdam: IOS Press. (Frontiers in Artificial Intelligence and Applications; 291)

II Pal, Deepak; Vain, Jüri (2019). Model based test framework for communications-critical internet of things systems. Databases and Information Systems X: Selected Papers from the Thirteenth International Baltic Conference, DB&IS 2018. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Amsterdam: IOS Press, 79-94

III Pal, Deepak; Vain, Jüri (2019). A systematic approach on model refinement and regression testing of real-time distributed systems. 9th IFAC Conference on Manufacturing Modelling, Management and Control, MIM 2019 : Berlin, Germany, 28-30 August 2019, Proceedings. Ed. Ivanov, Dmitry; Dolgui, Alexandre; Yalaoui, Farouk. Elsevier, 1091-1096

IV Pal, Deepak; Vain Jüri; Srinivasan, Seshadhri; Ramaswamy, Srini (2017). Model-based maintenance scheduling in flexible modular automation systems. 22nd IEEE International Conference on Emerging Technologies and Factory Automation, EFTA' 2017 : September 12-15, 2017, Limassol, Cyprus, 1-6

V Pal, D.; Vain, J. (2016). Generating optimal test cases for real-time systems using DIVINE model checker. BEC 2016 : 15th Biennial Baltic Electronics Conference, Tallinn University of Technology, October 3-5, 2016 Tallinn, Estonia, 99–102

VI Muthukumar, N.; Srinivasan, Seshadhri; Ramkumar, K.; Pal, Deepak; Vain, Jüri; Ramaswamy, Srini (2019). A model-based approach for design and verification of Industrial Internet of Things. Future Generation Computer Systems, 354–363

Abbreviations

RTDS	Real-Time Distributed Systems
SUT	System Under Test
MBT	Model-Based Testing
IoT	Internet of Things
IOTS	Input Output Transition Systems
TIOA	Timed Input Output Automata
IOCO	Input Output Conformance
CPS	Cyber Physical Systems
FSM	Finite State Machine
ISO	International Organization for Standardication
DTRON	Distributed Testing RealTime Systems Online
TA	Timed Automata
UTA	Uppaal Timed Automata
TIOCO	Timed Input Output Conformance
TTraces	Timed Traces
BEMS	Buildings Energy Management Systems
AGC	Automatic Gain Controllers
ROCOF	Rate of Frequency Change

1 Introduction

1.1 Chapter Overview

This thesis is about testing distributed real-time systems but as a background and to set the scene we begin with a description of the peculiarities pertaining to testing of real-time system in the context of model-based testing and why their testing is such a dominant factor in the software life-cycles. We introduce “What do *timeliness*, *latency*, *observability*, *controllability*, *reproducibility* and *non-determinism*” mean in the context of testing real-time systems.

1.2 Research Background

Real-time computer-based systems are woven into the fabric of our lives. They are often embedded, distributed systems and involve heterogeneous things where millions of sensors, actuators, and different devices interact with intelligent software.

Aeronautics, astronautics, medical devices, nuclear power generation and research, transportation, etc are the traditional examples of real-time safety-critical software systems. When employed in such systems, software is often responsible for controlling the behavior of electro-mechanical components and monitoring their interactions in addition to tasks such as user interface management, computer administration, and others. Since most accidents arise due to under-specified and/or some mismatch of the interfaces and interactions among the components, software correctness plays a direct and important role in system consistency and safety. The failure of such applications might cause significant and possibly dramatic consequences, such as interruption of public services, significant business losses, and including the loss of life.

The successful design of real-time systems is difficult and demands significant attention to detail. Such systems are usually embedded and distributed because they have timing deadlines that cannot be missed. Due to these complexities the development is a very specialized, expensive, methodical, slow, and process-driven field of software development. In most companies, software developers are primarily concerned with getting the software to "work", then going faster, shipping more features, and delivering more value. But software developers involved in the development of real time systems must be concerned primarily with creating safe systems which are designed, built, and tested to ensure it has ultra-low defect rates and ultra-high dependability. These applications require not only high availability, reliability, safety, and security but also regulatory compliance, scalability, and serviceability.

Despite being all around us, safety-critical software is not on the average developer’s radar. But recent failures of *Boeing’s two 737 Max crashes [60], Starliner test flight [61] etc.* have brought one of these companies and their software development practices to the attention of the public.

System must be fail safe. That is one of the fundamental principles of real-time safety-critical software systems design. This means that under any reasonable scenario where the system is being used in accordance with the operating instructions, it must not cause a dangerous situation if something goes wrong. For many systems that means stopping all actuators and reporting an error. For example, the blade in the food processor will stop immediately if the lid is removed while it is spinning. That is a simple case but for other systems, just figuring out how to fail safely is really difficult.

Software quality assurance does not start and surely does not end with testing. Because testing is applied to the final products of a development phase, defect discovery through testing always happens too late in each phase of product development. As per

software development/tesing life cycle, the common practice shows that defect prevention activities (by applying the appropriate constructive software engineering methods during product development in all phases) is more productive than any analytic quality assurance at the end of the development process. Nevertheless, testing is the ultimate sentinel of a quality assurance system before a product reaches the next phase in its life cycle. Nothing can replace good, effective testing in the validation phase before the product leaves R&D to go to manufacturing (and to customers). Even if this is the only and unique test cycle in this phase (if the defect prevention activities produced an error-free product, which is still a vision), it has to be prepared very carefully and be well documented. This is especially true for safety-critical software, for which, in addition to functionality, the effectiveness of all safeguards under all possible failure conditions has to be tested. There are few significant approaches to achieving reliability in a safety critical system:

- *Testing*. Testing is the process of identifying defects, where a defect is any variance between actual and expected results. Testing of real-time systems is the process of executing it to determine whether it matches its specification and operates properly in its intended environment. The continuous growth of systems complexity and high demand of security and reliability in the real-time systems has made their testing a big challenge. Moreover, majority of testing and verification techniques have been developed for the non-real-time systems and they cannot be applied on real-time systems due to timing constrains and concurrency issues.

- *Formal Specification and Verification*. For analyzing real-time systems, designers and developers have frequently used formal verifications techniques during design and development phase of systems [41, 42]. Verification is the act of proving the correctness of systems by determining that a system or module meets its specification. Formal methods [10] may be used to give a description of the system to be developed, at whatever level(s) of detail desired. In practice, rigorous mathematical proof at the code level is only suitable for small systems due to the state space growth that is exponential in the number of parallel components. Regardless the usage of several state space reduction techniques such as partial order reduction [43] and symbolic model checking [44, 45] the problem of scalability still prevents the verification of large-scale real-time systems.

- *Automatic Program Synthesis*. Program synthesis is a special form of automatic programming that is most often paired with a technique of formal verification. The goal is to construct automatically a program that provably satisfies a given high level specification. In contrast to other automatic programming techniques, the specifications are usually non-algorithmic statements of an appropriate logical calculus [66].

Modern large scale real-time systems have grown to the size of global geographic distribution and their latency requirements are measured in microseconds or even nanoseconds. Such applications where latency is one of the primary design considerations are called *low-latency systems* and where it is of critical importance – to *time critical systems*. Since large scale systems are mostly distributed systems (by distributed systems we mean the systems where computations are performed on multiple networked computers that communicate and coordinate their actions by passing messages), their latency dynamics is influenced by many technical and non-technical factors. Just to name a few, energy consumption profile look up time (few milliseconds) may depend on the load profile, messaging middleware and the networking stacks of operating systems. Similarly, due to cache miss, the caching time can grow from microseconds to about hundred milliseconds [62].

In-practice, testing time critical distributed systems is a difficult and challenging task. The spatial distribution, communication and presence of numerous components introduce timing imperfections which, if not considered during the design and deployment phases can lead to catastrophic outcomes that affects the reliability of the system as a whole. Building a reliable time critical systems requires a system to be tested for performance in the presence of these timing imperfections induced by various components. For instance, if in distributed systems, the timing latencies of the message communication or the occurrence order of the events is unknown it can make testing intractable. These challenging issues may not be suitably addressed by traditional centralized remote testing. Such challenges emerge also due to severe timing constraints, the tests have to satisfy when the required reaction time of the tester ranges near the message propagation time. These challenges restrict the capability of centralized remote testing [54] which cannot guarantee the controllability of distributed events, and respect the strict timing constraints. Reaching sufficient test coverage by integration testing of such systems in the presence of numerous latency factors and their interdependency, is out of the reach of manual off-line testing as well. Since, off-line testing of such systems is not possible due to the non-deterministic nature of system under test (SUT), off-line testing approaches need to be replaced by on-line distributed testing.

The need for automated online test generation and their correctness assurance have given rise to the use of model-based testing (MBT) and the development of several commercial and academic MBT tools [59]. For instance, smart connected factories with IoT based control systems is a new technology in manufacturing industries which undergoes frequent change in requirement specifications and tools, expecting reduction in testing efforts and costs [55]. In this context, MBT offers an automated tool support (regression testing) and platform independence thus aiming to lower the testing effort of IoT [63]. We interpret MBT in the standard way, i.e. as conformance testing that compares the expected behaviors described by the system model with the observed behaviors of an actual implementation [56, 57].

1.3 Testing Real-Time Software Systems

A real-time distributed systems (RTDS) are computer systems which typically interact with sub-systems and processes in the real-physical world. Such systems are in strong interaction with their physical environment and must respond to externally generated input stimuli within a finite and specified period. For example, the environment of a real-time system that controls a robot arm includes items coming down a conveyor belt and messages from other robot control systems along the same production line. They are also called *reactive systems* as they react to changes in their environment. These changes are recognized by sensors, and the system influences the environment through actuators. Since real-time systems control hardware that interacts closely with entities and people in the real world, they often need to be dependable. They have explicit time constraints that specify the response time and temporal behavior of real-time systems. For example, a time constraint for a flight monitoring system can be that once landing permission is requested, a response must be provided within 30 seconds. A time constraint on the response time of a request is called a deadline or reaction time. Time constraints come from the dynamic characteristics of the environment (movement, acceleration, etc.) or from design and safety decisions imposed by a system implementation.

Definition 1.1. *(Timeliness)*: It refers to the ability of a real-time system to meet timing constraints. It can be measured as the time interval from moment when system receives data, processes it, and returns results in right time as expected

Faults in the software can lead to software timeliness violations and costly accidents. Thus testers need to detect violation of timing constraints. Real-time systems usually are in strong interaction with its software and hardware components needed to make the system behave as intended. Real-time systems are defined by a set of tasks that implements a specific functionality of the system. Such tasks are defined in two types i.e. *Periodic* and *Dynamic*.

- *Periodic tasks*, are activated at regular intervals. A task repeats itself after a fixed time interval. A periodic task is specified by four tuples: $T_i = < \phi_i, P_i, e_i, D_i >$ where, ϕ_i is the task's first release time. If it is not specified then release time of first task is assumed to be zero; P_i is the period of the task, i.e. the time interval between the release times of two consecutive task activations; e_i is the execution time of the task; D_i is the relative deadline of the task. For example, consider the task T_i with period = 5 and execution time = 3. The task is released first time at t = 0 (assumed) then it executes for 3s and then the task is released again at t = 5 which executes for 3s and then next time again at t = 10. So the task is released at t = 5k where k = 0, 1, . . ., n. Since tasks are released at fixed-interval, all the intervals in time when such tasks are activated are known beforehand.

- *Dynamic tasks* are further divided into two categories based on their criticality and knowledge about their occurrence times.
  —*aperiodic tasks* where tasks are released at arbitrary time intervals, i.e. randomly. Aperiodic tasks have soft deadlines or no deadlines so that processes can be finished after its deadline, although the value provided by completion may degrade with time.
  —*sporadic tasks*, they are similar to aperiodic tasks, i.e. they repeat at random instances. The only difference is that sporadic tasks have hard deadlines. To achieve timeliness in a real-time system, aperiodic tasks must be specified with constraints on their activation pattern. When such a constraint is present, the tasks are called sporadic. A common constraint is a minimum inter-arrival time between two consecutive task activations. Tasks may also have an offset that denotes the time before any instance may be activated.

Challenges with testing real-time systems: Testing RTDS is inherently more complex than testing untimed systems, mainly because of communication and synchronization requirements between cooperating components/processes, whose actions may depend on actions of other components and on the time and duration of these actions. Testing functional correctness is not just only objective of testing RTDS, rather ensuring the timeliness of results produced is equally important. In the context of testing real-time systems following aspects of testability have been identified:

- *Observability*: It is the ability to monitor the behavior of system under test (`SUT`). It is important to determine whether the `SUT` performs as expected. Most important aspect to understand observability is that tester must be able to observe every single event (actions) generated by its environment and most importantly to determine correct ordering and timing of events without *probe-effect* (the delays introduced by insertion or removal of code instrumentation may result in unpredictable behavior). Observability can be achieved by injecting monitors (probes) into `SUT`

to record relevant events (timing and the order of input/output events at different ports) and log the time stamps for global monitoring. The monitored data is collected and integrated to obtain a coherent view of the system. These monitors can be called by applying input to SUT from the location where monitors are placed. However, generating and deploying the monitors for a complex distributed application is a significant engineering effort. Modifying existing distributed systems by instrumenting the monitors may introduce delays and network overhead (probe effect) but there has been lot of research on the implementation of monitors with the aim of obtaining a coherent view of the system and achieving this in a non-invasive manner [51, 52, 53].

- *Reproducibility & Controllability*: Tester must be able to control the test execution over SUT to establish the test preconditions, create test inputs and most importantly, reproduce arbitrary test scenarios. Reproducibility is when the system repeatedly exhibits identical behavior when stimulated with the same inputs. It is a very desirable property for testing, particularly during regression testing and debugging. Achieving reproducibility in RTDS, is difficult due to nondeterministic behviour of SUT. It is therefore important to have possibly high degree of controllability to effectively test non-deterministic behavior of SUT.

- *Non-Determinism*: Increasing reliance on real-time systems increases nondeterministic environmental inputs (e.g., from sensors) and from communication delays (e.g., in cloud and edge computing). Nondeterminism occurs when there is no conceptual way of pre-determining the SUT exact behavior. This can be due to overwhelming complexity or to inadequate observability and controllability. It has serious impact on testability, for example running same tests (*input-actions*) multiple times with exact same *pre-conditions* can produce distinct results (i.e., different output-actions and post-conditions). Non-determinism can have different reasons and forms: *(i) concurrent non-determinism* (due to system internal and external concurrency); *(ii) emergent non-determinism* (due to integration of subsystems into systems); *(iii) physical non-determinism* (due to the nature of physical processes); *(iv) exceptional non-determinism* (due to fault and failure behavior).

Testing for Timeliness: Majority of testing and verification techniques have been developed for non-real-time systems and they cannot be applied for real-time systems due to timing constrains and concurrency issues. Timing is traditionally analyzed and maintained using scheduling analysis techniques. Full schedulability analysis of real-time systems with lack of observability, controllability and determinism is complicated and can result in undetecting timeliness violations in the presence of timing faults. Therefore, the challenge in testing is to find timed test sequences execution orders that will cause timeliness faults that possibly result in failure. In the past, several model-based testing methodologies have been proposed where testing activities are guided by the formal model of system. For example, models formalized in Real-Time UML, timed petri nets, timed automata have been used as test oracles and to define coverage criteria such as edge, condition or path coverage to check the violation of timeliness constraints. Having such formal models available the test designers and developers have combined testing with formal verification techniques during design and development phase of systems [41, 42]. In practice, rigorous mathematical proof at the low level of abstraction in the model is only suitable for small systems due to the state space growth that is exponential in the number of parallel components. Regardless the usage of several state space reduction techniques such as partial

order reduction [43] and symbolic model checking [44, 45] the problem of scalability still prevents testing and verification of large-scale RTDS.

In this thesis, we consider a RTDS, where a SUT is abstracted in the model up to its observable behavior at test ports and related timing constraints are posed on input/output events. Regardless the high level of abstraction the need for testing timing correctness restricts the relevance of centralized remote testing methods which are commonly used in RTDS testing. This is because a centralized test architecture cannot guarantee the controllability of distributed input injection, and respect their timing constraints, namely, message propagation time between the tester and SUT becomes the limiting factor. Another aspect to be considered in RTDS is that reaching sufficient test coverage by integration testing of such systems in the presence of numerous latency factors and their interdependency, is out of the reach of off-line testing. Since, off-line testing of such systems is not possible due to the non-deterministic nature of SUT off-line testing approaches need to be replaced by on-line distributed testing. The need for automated online test generation and tests correctness assurance have given rise to the combined use of MBT and distributed testing. We interpret MBT in the standard way, i.e. as input/output conformance (ioco) testing [56] that compares the expected behaviors described by the system model with the observed behaviors of an actual implementation. As stressed above, due to inherent non-determinism of distributed systems the natural choice is online MBT where the test model is executed in lock step with the SUT. The communication between the model and the SUT involves controllable inputs of the SUTand observable outputs of the SUT which are required to be conforming with the requirements model for detecting errors in SUT.

1.4 Current Practice and Related Work

The continuous growth of systems complexity and high demand of security and reliability in the RTDS has made their testing a big challenge. Moreover, majority of testing and verification techniques have been developed for non-real-time systems and they cannot be applied on real-time systems because of not addressing the timing constrains and concurrency issues. Testing RTDS requires an integration of computation, communication and control in the test architecture. In the current practice, overall system functionality is tested for conformance with requirements. The main objective of conformance testing is to determine whether a system complies with the requirements of a specification where inputs are applied to observable ports of system and outputs from observable ports are analyzed against expected outputs.

In the current practice, centralized remote testing methodologies [11] used to test RTDS apply test architecture as shown in Figure 1. The remote tester generates test inputs for the distributed ports, waits for the responses, and continues the process with the next set of inputs. Such a remote testing approach works when the communication and timing constraints of the SUT are negligible compared to SUT reaction time. However, in most distributed applications, the timing constraints are significant and time-varying thereby introducing non-determinism. This leads to situations wherein the remote tester is unable to generate the necessary inputs for the SUT within expected time window, thereby making the test cases incomplete in terms of timing requirements coverage [27]. Major challenges emerge due to several reasons:

- *severe timing constraints*: the tests have to perform fast enough to mimic load profiles of the real environment. It is critical when the required reaction time of the tester ranges near the message propagation time;

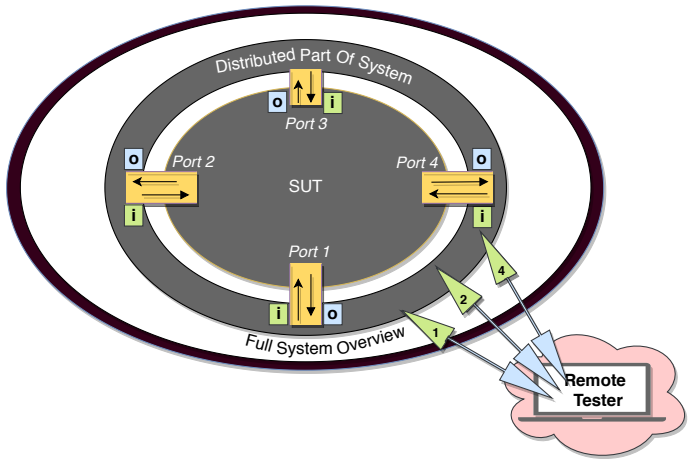


Figure 1 – Traditional approach for testing distributed systems

- *order constraints*: due to non-deterministic nature of message propagation delays in distributed systems, maintaining same order at receiving as the messages are sent is hard to guarantee.

- *test repeatability*: since remote tester cannot control the simultaneous inputs at different remote ports and observe the outputs of RTDS at the same time, the possibility of i/o actions interleaving cannot be fully eliminated and due to this non-determinism the test repeatability is not possible;

These problems restrict the usability of centralized remote testing which has limited capability of controlling distributed events, and respecting the timing constraints.
An alternative to the remote testing is a *distributed testing*. In distributed testing architecture, shown in Figure 2, the remote tester is decomposed into a set of communicating local testers. This modification eliminates the message propagation time between the local tester and SUT as the testers are attached directly to the ports enabling simultaneous test inputs and it reduces the overall test response time.
Most of the modern cyber-physical systems (CPS) such as modern transportation systems, energy grids etc., are large scale systems that have a massive distributed architecture. While testing such systems in itself is a challenge, their strict real-time requirements complicate the testing further. Current practice to use manual testing is error prone and expensive. Therefore, automating the testing process has received considerable attention in recent years and the model based testing (MBT) has emerged as a promising approach. Typically a MBT uses the model which either is the specification of the system under test (SUT) or specifies some behavioural aspect that is intended to be tested. Recent results have shown that MBT can be used to test complex industrial automation systems requiring strict real-time behaviours [28, 29] and cyber-physical systems with hybrid dynamics[30]. The MBT approaches in the literature use either state-based or behavioural models. Their semantics are described mostly by the Finite State Machines (FSMs) or input-output transition systems (IOTS) including additional information, e.g. time [31] -[32]. However, adopting these models to emergent applications (e.g., smart grids) is becoming a challenge due to two reasons: *(i)* scale, distribution and interconnections of the systems, and *(ii)* real-time requirements.

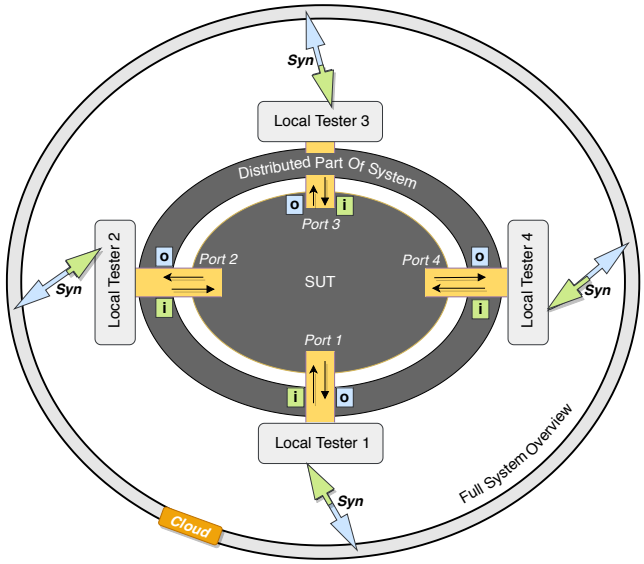


Figure 2 – Distributed Test Architecture

Testing distributed systems has been one of the MBT challenges since the beginning of the 90s. An attempt to standardize the test interfaces for distributed testing was made in ISO OSI Conformance Testing Methodology [33]. A general distributed test architecture, containing distributed interfaces, has been presented in Open Distributed Processing (ODP) Basic Reference Model (BRM), which is a generalized version of ISO distributed test architecture.

As for broader context of distributed testing the early works focused on testing distributed non real-time systems [36, 64, 40]. The theory of testing distributed real-time systems has gained interest only in recent years when global time keeping techniques emerged [65]. First MBT approaches represented the test configurations as systems that can be modeled by finite state machines (FSM) with several distributed interfaces, called ports. An example of abstract distributed test architecture is proposed in [23]. This architecture suggests that `SUT` contains several ports that can be located physically far from each other. The testers are located in these nodes that have direct access to ports. There are also two strongly limiting assumptions: (i) the testers cannot communicate and synchronize with one another unless they communicate through the `SUT`, and (ii) no global clock is available. In practice there is no global clock that can synchronize the testers. These ideas led to the ISO standard on distributed testing architecture [33]. The architecture lacked mechanism to coordinate the local testers and the concept of global clock.

Works studying controllability and observability of distributed testing were studied in [35]. These notions were used in [34] to investigate fault detectability and synchronization. In contrast, the authors in [36] proposed a coordination algorithm that allows testers to exchange information employing reliable communication channels. With the introduction of the extra communication messages, undesirable delays and overheads were also created. To overcome the difficulties with such works, the use of offline test suites to guarantee controllability and observability was proposed in [37]. However, generating test sequences for non-deterministic `SUT` offline to guarantee test controllability and observability is seldom possible. It is therefore, imperative to use online distributed testing as in [64]. The use of online testing approach to solve controllability and observability prob-

lems for distributed testing was proposed in [40]. The investigation further showed that even in the absence of critical timing constraints, the SUT should satisfy some constraints to have controllability and observability problem decidable.

These problems occur if a tester cannot determine either when to apply a particular input to SUT, or whether a particular output from SUT is generated in response to a specific input [37]. For instance, the controllability problem occurs when the tester at a port p_i is expected to send an input to SUT after SUT has responded to an input from the tester at some other port p_j, without sending an output to p_i. The tester at p_i is unable to decide whether SUT has received that input and so cannot know when to send its input. Similarly, the observability problem occurs when the tester at some port p_i is expected to receive an output from SUT in response to a given input at some port other than p_i and is unable to determine when to start and stop waiting. Such observability problems can introduce fault masking.

In [37], it is proposed to construct test sequences that cause no controllability and observability problems during their application. Unfortunately, offline generation of test sequences is not always applicable. For instance, when the model of SUT is non-deterministic it needs instead of fixed test sequences online testers capable of handling non-deterministic behavior of SUT. But even this is not always possible. An alternative is to construct testers that includes external coordination messages. However, that creates communication overhead and possibly the delay introduced by the sending of each message. Finding an acceptable amount of coordination messages depends on timing constraints and finally amounts to finding a tradeoff between the controllability, observability and the cost of sending external coordination messages.

The need for retaining the timing and latency properties of testers became crucial natively when time critical cyber physical and low-latency systems were tested. Recently, results on testing timing correctness of remote testers were proposed in [38]. The investigation considered a SUT with distributed ports that are remotely observable and controllable, then the 2Δ condition was used for proving the tester's timing correctness. Here, the 2Δ describes the communication delay upper bound requirement between the SUT and remote tester as sufficient condition for the test controllability. Though this approach works reasonably well for systems with sufficient timing margins, but cannot be extended to systems with the timing constraint close to 2Δ or below. This means that the messages may not reach the port in time and as a result, the testing becomes infeasible in such systems.

More recently, the use of distributed testing for testing deterministic SUT with multiple ports was studied in [58]. The approach addresses the *Controllability Problems* of the distributed test architecture, when there is not enough information at the local ports to coordinate the input sequence with its peers. It is shown under what conditions the partial observability of SUT leads to situations where the distributed test inputs cannot be coordinated correctly.

1.4.1 Challenges in Centralized Remote Testing

Impact of latency in remote testing: A centralized test architecture introduces challenges when there is significant latency compared to reaction time requirements, which means messages are not always received in the order they are sent. In order to manage message propagation delay between tester and the SUT, tester should not wait for each output before sending input to SUT. However, to mimic realistic i/o traffic the test inputs to SUT

have to be sent simultaneously or following timing patterns that cannot be reproduced by linear stimulus-response sequences.

Example 1.1. Consider the timed i/o automata specification Spec in Figure 8 (a). Assume that the propagation latency between SUT and Spec is exactly 3 time units, which means if Spec has to apply input to SUT, it should send that input 3 time units earlier as demonstrated in Figure 3 , so that SUT receives the input on time as specified in specification. To maintain the propagation delay, the SUT and tester shall observe the timed trace as follows:

$\rho_{\mathtt{SUT}}$:	$(5 \cdot in[1]!) \cdot (7 \cdot out[3]?) \cdot (11 \cdot out[2]?) \cdot (12 \cdot in[1]!) \cdot (15 \cdot out[2]?)$
$\rho_{\mathtt{Spec}}$:	$(2 \cdot in[1]!) \cdot (9 \cdot in[1]!) \cdot (10 \cdot out[3]?) \cdot (14 \cdot out[2]?) \cdot (18 \cdot out[2]?)$

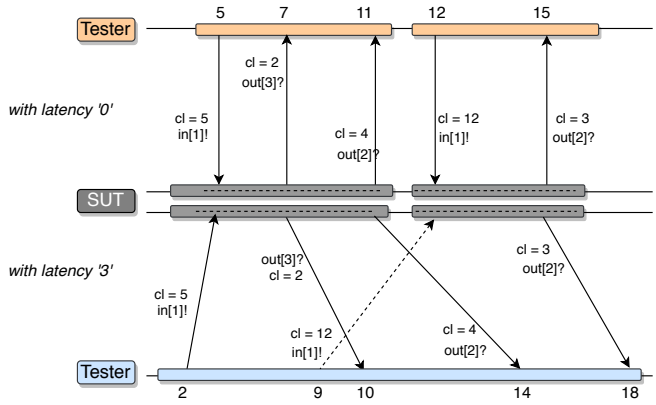


Figure 3 – SUT and Centralized Tester Communication with latency

As seen in $\rho_{\mathtt{Spec}}$, the Spec sends second input $in[1]!$ at 9 time units before receiving outputs $out[3]?$, and $out[2]?$ in response to previous input $in[1]!$ at 2 time units to SUT. It seems, outputs $out[3]?$, and $out[2]?$ are generated in response to second input $in[1]!$, though SUT produces outputs as specified in Spec and sends $out[3]?$, and $out[2]?$ to Spec before receiving the second input $in[1]!$. However, the emission of second input $in[1]!$ depends on the reception of an outputs $out[3]?$, and $out[2]?$, because of latency and to maintain it, tester should not wait to receive outputs before sending the input to SUT. This means in remote testing the propagation latency between SUT and Spec may lead to interleaving of input/output actions. This affects the generation of inputs for the SUT and the observation of outputs that may trigger a wrong test verdict.

Consider the remote testing architecture depicted in Figure 1 and its corresponding input-output transition system (IOTS) model in Figure 8. The SUT has 3 ports (p_1, p_2, p_3) in geographically different places with inputs/outputs $in[1]/out[1]$, $in[2]/out[2]$ and $in[3]/out[3]$ at ports $p1$, p_2 and p_3 respectively. The model defines the expected global behavior of SUT. Each symbolic input and output is expressed as the label of some edge of the model.

To model simultaneous actions at different ports the edges with multiple simultaneous communication actions are split into a series of transitions each carrying exactly one i/o action label and connected with auxiliary states labeled with "c". This notation is to specify that all ports of simultaneous i/o actions such group are updated at the same time. Using above modelling pattern, we model a three port automata shown in Figure 4 and

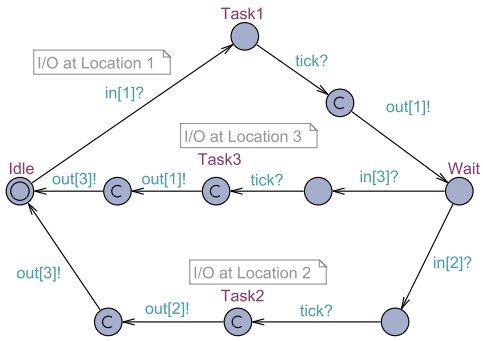


Figure 4 – SUT Model

5 where the tester sends an input $in[1]$ to the port p_1 at `Geographic Place 1` and receives a response or outputs $out[1]$ from `SUT` at same port. After receiving the output, the tester automaton is in state `Wait`, it gets both $in[3]$ on port p_3 and $in[2]$ on port p_2. Then, either it follows the intended path sending $in[3]$ before $in[2]$, or it sends $in[2]$ before $in[3]$. If tester decides to send $in[3]$ before $in[2]$ it receives an output $out[3]$, $out[1]$ at port p_3 and p_1 respectively and returns to state `Idle`. If tester decides to send $in[2]$, the `SUT` responds with an outputs $out[2]$, $out[3]$ at port p_2 and p_3 respectively. Similarly, based on previously triggered inputs and received outputs the next input is sent to `SUT` and tester continues with the next set of inputs and outputs until the test scenario will be finished. Suppose the described `SUT` is a real-time distributed system, which means that it has strict timing constraints for messaging between ports. More specifically, after sending the first input $in[1]$ to port p_1 at `Geographic Place 1` and after receiving the response $out[1]$, the tester needs to decide and send the next input $in[2]$ to port p_2 at `Geographic Place 2` or input $in[3]$ to port p_3 at `Geographic Place 3` in Δ time. Since `SUT` is distributed in a way that signal propagation time is non-negligible, this can lead into a situation where the tester is unable to generate the necessary input for the `SUT` with in expected timing window due to message propagation latency.

Another important aspect that needs to be addressed in remote testing is *functional non-determinism* of the `SUT` behaviour with respect to test inputs. For non-deterministic systems only online testing (generating test stimuli on-the fly) is applicable in contrast to that of deterministic systems where test sequences can be generated offline. Second source of nondeterminism in remote testing of real-time systems is communication latency between the tester and the `SUT` that may lead to *interleaving of inputs and outputs* discussed in *Example 1.1*.

Consequently, the centralized remote testing approach is not suitable for testing a real-time distributed system if the system has strict timing constraints that require reactions faster than 2Δ. The shortcomings of the centralized remote testing approach are mitigated with extending the Δ-testing idea by partitioning the monolithic remote tester into multiple local testers as shown in Figure 2.

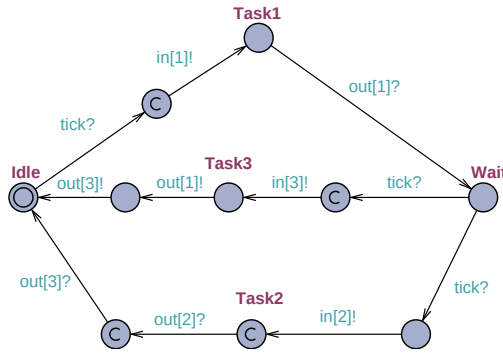


Figure 5 – Remote Tester Model

1.5 Objectives of Thesis

The need for model-based testing methodologies in the context of real-time systems has been recognized commonly by academic and industry world. However, very few of them adapted MBT as standard testing techniques. To meet the demand and advance the theory of distributed testing, the Thesis research has following objectives:

- Development of model-based distributed testing methodology, a practical and provably correct distributed test architecture for time critical distributed systems possibly with non-deterministic behavior.
- Showing the relevance of Uppaal timed automata (UTA)[1] formalism for specifying the behavior of real-time distributed systems. It includes
  - Introducing semantic assumptions for modelling *n-ports* distributed systems under test.
  - Showing how a multiple-port distributed systems timing constraints can be modeled using UTA constructs such as urgent and committed locations, clock invariant, clock guards etc.
  - Analyzing what is needed to generate distributed testers from those models.
- As constructive part of the thesis, the objective is to create a method and algorithmic implementation of that for generating distributed communicating tester models from centralized remote tester models using partitioning algorithms and SUT configuration specification over geographical locations.
- Showing that the distributed tester generated from the centralized remote tester not only preserves the correctness of the centralized tester but also mitigates the testers reaction time, i.e., the distributed testers meets (one) Δ controllability requirement against 2Δ of the remote tester.
- Validating the proposed MBT approach on an industry scale case study by applying it for functional and performance testing of flexibility contract protocol for Ancillary Services in Energy Grids.

[1] http://uppaal.org

1.6 Thesis Contributions

The key task of the research and novelty lies in the elaboration of fully automated test generation technique for distributed online testing and its implementation in the form of time-efficient algorithms. To achieve the objectives of this thesis, the main contributions are:

- An efficient online model-based test architecture for real-time distributed system models (with non-deterministic behavior) that satisfies the timing constraints for solving controllability and observability issues of centralized testing architecture.

- A test scenario where a centralized testing cannot be applied. It is shown experimentally that the distributed test architecture is more scalable and efficient in terms of test reaction time-wise than centralized remote test architecture for testing large number of geographical locations (ports) in a system.

- A semantic foundation for modelling *n-ports* distributed systems using Uppaal Timed Automata.

- Partitioning algorithms to produce distributed local testers from given remote tester model. The generated testers not only preserves the functional correctness of the centralized remote tester but also mitigate the testing time, i.e. the distributed testers meets (one) Δ controllability requirement against 2Δ of the remote tester where Δ is the granted message end-to end propagation time.

- Bisimulation based verification method for proving the correctness of test distribution algorithm.

- Support for regression testing, a systematic model refinement approach where alternative implementation configurations can be extracted from an abstract test model. We demonstrate that testing by partitioning a monolithic tester model into a set of distributed communicating testers makes model based testing for manufacturing automation systems scalable.

- Demonstration of the proposed approach on a case-study using MBT tool DTRON as an available test execution platform that is practically usable for facilitating distributed testers deployment and management.

1.7 Thesis Structure

The rest of the thesis is structured as follows:

`Chapter 2`: We present the preliminaries of this thesis and formal definitions related to testing of real-time distributed system in the context of model-based testing. We discuss the major issues in testing distributed testing by illustrating it with examples. We analyze the features of online model-based testing and its advantages compared to manual/offline black-box testing. We show how a multiple-port distributed real-time system models can be build with UPPAAL modelling formalism. We compare different aspects of testing centralized, decentralized and distributed systems. We present a related work on testing distributed systems and discuss different testing approaches. We present a motivation to adapt distributed test architecture over centralized testing. Finally, we introduce

the concept of timed input/output conformance relation which is used to capture the conformance of SUT to its specifications.

Chapter 3: This chapter present the main results of thesis. We present the test architecture for testing distribute system and discuss its components such as adapter, coordination between adapter and local testers. We present several hypothesis and definitions required to present our approach. We elaborate the issues in distributed testing with examples. We introduce the test distribution algorithms and compare their performances.

Chapter 4: The feasibility of approach is demonstrated on case study "Flexibility Contracts for Ancillary Services in Energy Grids". We demonstrate the proposed test approach by using MBT platform DTRON as a test execution platform.

Chapter 5: Presents the conclusions and outlines the directions for future research.

1.8 Chapter Summary

In this chapter, we have described the peculiarities pertaining to testing of real-time distributed system in the context of model-based testing and why their testing is such a dominant factor in the software life-cycles. We have discussed “what do *timeliness*, *latency*, *observability*, *controllability*, *reproducibility* and *non-determinism* ”mean in the context of testing real-time systems. We have also discussed the significant approaches such as *testing*, *formal specification and verification* and *automatic program synthesis* to achieve reliability in real-time systems. We have discussed the challenges and need for automated online test generation techniques for testing real-time distributed systems. We have presented a related work on testing distributed systems. Finally, we finished the chapter by discussing the research objectives and presented novel contributions. We identified the importance of requirements for the development of efficient testing methodologies for real-time distributed systems with the aim of automation, keeping in mind constraints imposed by academic and industry needs.

2 Preliminaries

2.1 Chapter Overview

We introduce the basics of online model-based testing and discuss the advantages compared to manual/offline black-box testing. We discuss the MBT taxonomy step by step during which we will refine these concepts in subsequent sections. We introduce a semantic foundation for modelling real-time systems with timed automata. We also discuss how the specifications are transformed from timed automata to more expressive formalism Uppaal timed automata (UTA) to exploit the power of Uppaal simulation, verification and testing tools. We introduce the *n-ports* timed automata and show how a multiple-port distributed real-time system test models can be build with UPPAAL modelling formalism by considering timing-constraints such as urgent and committed locations, clock invariant etc. Finally, we introduce the concept of timed input/output conformance relation which is used as test pass criteria confirming the conformance of SUT to its specifications.

2.2 Model-Based Testing

Model based testing (MBT) technologies have emerged as a set of potentially powerful methods and tools which have become a standard in recent years for modern test automation industry. MBT is a testing approach where test cases are automatically generated from models. The models specify the expected behavior of the SUT. The models are also used to describe the test environments and test strategies, generating test cases, symbolic test execution and for evaluating test design quality. A model is an abstraction of the real-world function. Typically, a formal model provides an unambiguous specification of desired SUT behavior which has clear syntactic representation and semantic meaning. These models are used to generate automatic test cases, and they represent how we expect the system to behave under test.

MBT is generally understood as black-box conformance testing where the models are used as specifications of required observable interactions between SUT and its environment [12]. The goal is to project the abstract behaviours described in the model onto SUT by sending model generated test stimuli to SUT and observing if reactions of SUT conform to those specified in the model. During testing, test execution environment runs selected test cases on the SUT and emits a test verdict (pass, fail, inconclusive). The verdict shows correctness in the sense of input-output conformance (ioco) relation between the requirements model and its implementation [56].

An important motivator of the MBT is to stimulate the testing process towards automation. In contrast with traditional manual black-box testing, MBT process can be fully automated by means of contemporary testing tools. MBT ensures the possibility to trace the correspondence between requirements, models and test cases. The advantages of MBT compared to the traditional approach are as follows:

- *High degree of automation*: Instead of writing hundreds of test scripts manually from specifications that is common practice in manual testing, MBT tools can be used to automatically generate test suite from the model, which saves design time and prevents making human mistakes.

- *Independent testing*: Models represent the requirements to SUT and are completely separated from its implementation. This allows avoiding carrying potential implementation errors in the test model.

- *Complex test cases*: Since models abstract from many implementation details they

allow to cover much larger fragments of SUT functionality. Such models are helpful in generating systematically unique and complex test cases.

- *Unambiguous specifications*: MBT helps to discover and eliminate ambiguous requirements. By focusing on important aspects the models improve human comprehension of requirements intention and keep them clean from inconsistences.

- *Easy maintenance*: Well-structured models improve traceability of requirements that need to be addressed in the test cases and enhance updates when test fails. Test fails refer to the non-conformance between the model and system implementation. Only corresponding model needs to be updated instead of writing all test again when there is change in specifications.

- *High test coverage*: MBT enables easy specification of the different model coverage criteria, such as all-states, all transitions, selected branching conditions etc. that can be used to extract the corresponding test cases. The advantage is that one can automatically generate a variety of test cases from the same model simply by choosing different test generation criteria.

Depending on how tests are generated and deployed MBT methods can be divided into two broad groups: *off-line* and *on-line* testing. In off-line testing the tests are generated before deploying and executing them. In on-line testing the test stimuli are decided and sent to test interface on-the-fly based on the SUT earlier reactions (output) and its current state. Regardless the generation methods the MBT process can be divided into five main steps as shown in Figure 6 [12].

1 *Building the model of* SUT *and its environment*: The key decision in building a model of SUT for testing is deciding an accurate level of abstraction, that is, deciding which functionality of SUT to include or disregard in the model. It is advisable to keep the model small in respect to the size of SUT but it must be detailed enough to specify test objectives. It is also preferable to build several small partial models than one complex model, i.e. building a model for each subsystem or design view and test them (if possible) independently. Building a large monolithic model for full system may end up with one of the most serious problems faced when applying formal verification on the model. This problem is generally known as "state explosion problem". Building a model by preserving two significant properties (limited size and sufficient level of detail) can be difficult at times. This is why transforming the accurate abstract behavior into a model has become an engineering challenge to decide what level of abstraction should be chosen for modelling to satisfy test objectives. Also, it presumes deciding on how much certain functionality can be neglected and most importantly which modeling notations can encode those abstract details naturally. Once we have a model ready, it is advisable to verify the model whether it is consistent and has the desired behavior. Before we start generating test suites from this model, it makes sense to find flaws up front because flawed requirements breed bugs that needs to get rid of later, often at a high cost to the project bottom line. Model checking has proven to be a successful method to verifying correctness of model with respect to temporal logic specifications. It allows to explore the behavior of the model exhaustively and verify its correctness against given requirements.

2 *Generating Abstract Tests from Model*: In principle, the models can generate an infinite number of possible test scenarios. Therefore, test selection criteria, specified

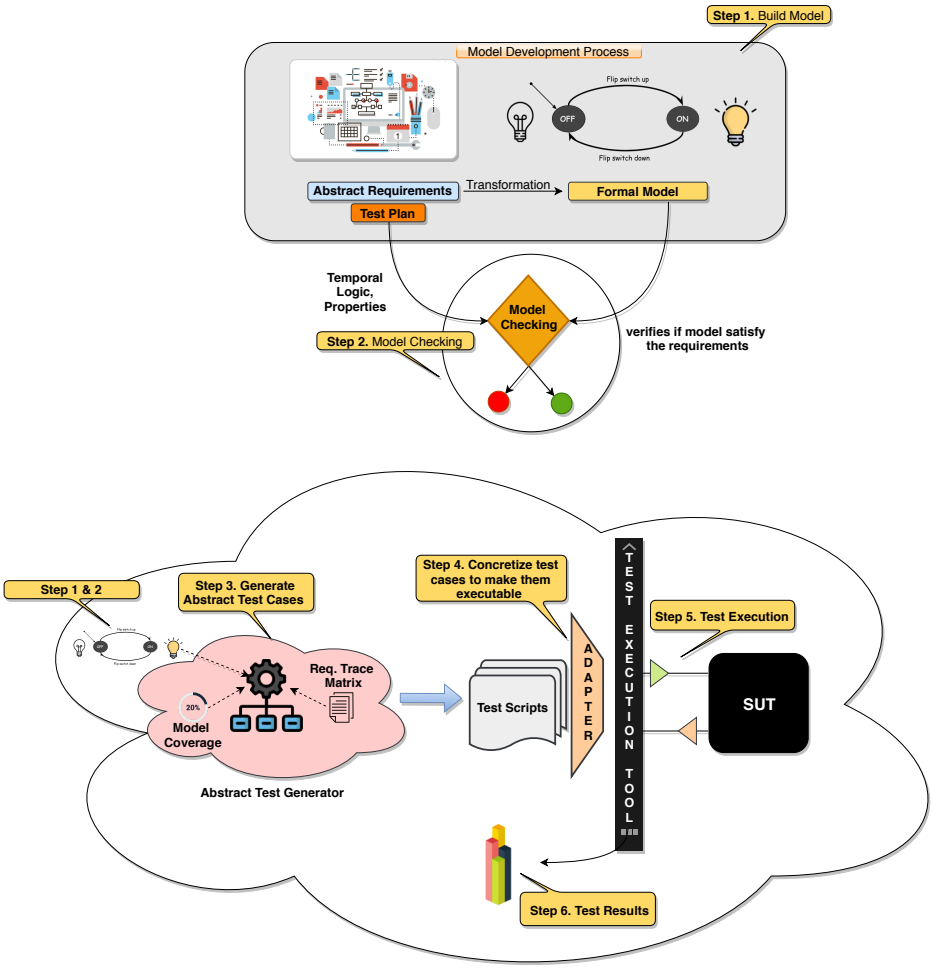


Figure 6 – Model-Based Testing Process

as test purpose are meant to select a finite and practically executable set of feasible test cases. For example, different model coverage criteria, such as all-states, all transitions, selected branching points etc. can be used to derive the corresponding test cases. Note that, abstract tests generated from the model are not directly executable since they are in the form of symbolic input-output action sequences and states from the abstract model. Most MBT tools also generate a coverage statistics or a requirement traceability matrix, which give some feedback about the quality of generated test set or indicate which part of the model is not investigated or tested well. For example, if some part of the model has not been explored for particular test purpose, we could try to change the test parameters and repeat the same generation step. Also, if we want to improve the coverage of the test one can add auxilliary conditions in the model that guide the test generation tool to find optimal coverage test sequences.

3 *Concretizing the abstract tests to make them executable*: Third step of MBT is to transform the abstract test suites into executable tests. The goal is to bridge the gap between abstract tests and the physical SUT by introducing low-level SUT details that were neglected in the abstract model. This can be achieved by encoding adapters which map symbolic model inputs to executable ones and the concrete outputs of SUT back to symbolic form to compare them with ones given in the model. An advantage of the separation between abstract test suite and concrete test suite is the platform and language independence of the abstract test cases. Thus the same abstract test case can be reused in different test execution environments. An alternative to the use of test adapters is the transformation of abstract tests to directly executable test scripts. Both approaches are in use but, as a rule, building adapters requires less effort than constructing script generators when the requirements to scripting format change.

4 *Executing the tests on* SUT *and assigning verdicts*: Next step is to execute the concrete tests against SUT. During testing, a tester executes selected test cases on the SUT and emits a test verdict (pass, fail, inconclusive). The verdict shows correctness in the sense of ioco [13, 14].

   Due to inherent non-determinism of distributed systems often the only option for integration testing is on-line testing. Online MBT is executed in lock step with the SUT. The communication between the model and the SUT involves controllable inputs of the SUT and observable outputs of the SUT which makes easy to detect ioco violations. In case of non-deterministic systems a single pre-computed (offline) test sequence may never reach the test goal, and instead of a sequence we need an online testing strategy that is capable of reaching the goal even when SUT provides nondeterministic responses to a test stimulus. The issue is addressed in [15] where the reactive planning online tester synthesis method is proposed.

5 *Analyzing the test results*: After test execution, given results are analyzed and corrective actions are taken in the implementation if needed. Hereby, for each test that reports a failure, the cause of the failure is determined and the program (or model) is corrected.

An example of the symbolic test execution tool for Uppaal Timed Automata is Uppaal Tron [52] which conceptual architecture is depicted in Figure 7. For detailed overview of MBT and related tools we refer to [12].

2.3 Modelling of Distributed Real-Time Systems

2.3.1 Timed Automata

To define the testing architecture and algorithms formally we need to introduce a semantic foundation for modelling the RTDS. We describe the notions of timed automata (TA) introduced by [48] as a formalism to model the behavior of real-time systems. TA is a formalism to annotate a transition system with timing constraints using finitely many real-valued clocks. This set of real-valued clock-variables track the time progress and can guard on transitions to restrict the behavior of automaton. We consider time domain $\mathbb{T}$ as set $\mathbb{R}_{\geq 0}$ of non-negative reals that valuate the variables called clocks denoted by $\mathbb{C}$.

Definition 2.1. *(Clock valuations, operations on clocks).* A clock valuation is the function $\mu : cl \to \mathbb{R}_{\geq 0}$ mapping a clock cl from the set of clocks $\mathbb{C}$ to the set of non-negative reals

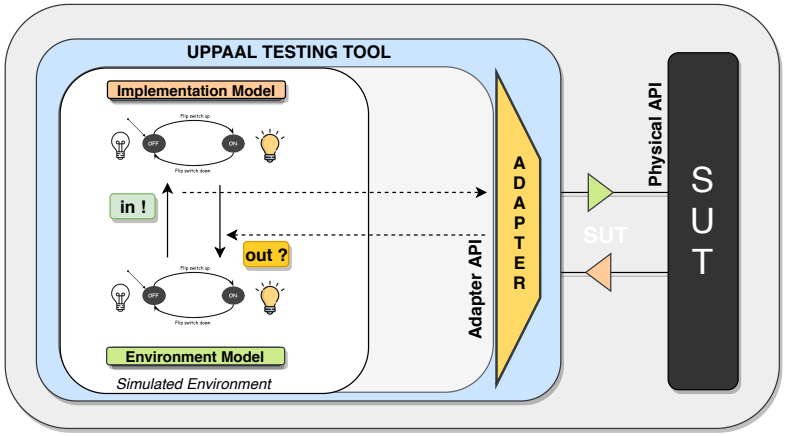


Figure 7 – Online MBT Deployment Process

$\mathbb{R}_{\geq 0}$. Let ϕ represents the set of guards on clocks, a conjunction of constraints of the form $cl \bowtie t$, and let U represents the set of updates of clocks corresponding to sequences of statements of the form $cl := t$, where cl $\in \mathbb{C}$, t $\in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$. Guards and invariants ($\mathbb{I}$) are considered as first-order logic expressions (with predicate symbols $\bowtie$) over clocks cl and time constants t. Notation $U \models \mathbb{I}(l)$ means that U satisfies invariant at location *l*.

Definition 2.2. *(Timed Automaton [48, 49, 50])*. A timed automaton (TA) is a tuple (L, l_0, $\mathbb{C}$, Σ, $\mathbb{I}$, E) where

- *L* is a finite set of locations (or nodes);
- $l_0 \in$ L is the initial location;
- $\mathbb{C}$ is finite set of clocks;
- $\mathbb{I} : L \rightarrow \phi(\mathbb{C})$ *assigns invariants on clocks to locations.*
- *E* is a finite set of edges between locations with an action, a guard and a set of clocks to be reset. Each edge is a tuple $(l, l^{'}, \phi, cl, d, a)$ where:
  - $l, l^{'} \in L$ are the source and destination locations;
  - ϕ is a guard, a conjunction of constraints of the form $cl \bowtie x$, where $cl \in \mathbb{C}$, x is an integer constant and $\bowtie \in \{<, \leq, =, \geq, >\}$;
  - $cl \subset \mathbb{C}$ is a set of clock to reset to zero;
  - $d \in \{Committed, Urgent, Normal\}$ is the type of location; note that in the modeling we use location types as in Uppaal TA introduced by Larsen et al [18];
  - $a \in \Sigma$ is a finite set of actions, co-actions and the internal τ-action.

A timed automaton defines (possibly infinite) timed labeled transition system (TLTS) and its states are pair s= (l, v), where $l \in L$ is a location and $v \in \mathbb{R}^{\mathbb{T}}_{\geq 0}$ is a clock valuation satisfying the invariant of location l. A transition in TA is denoted as $l \xrightarrow{\phi, a, cl} l'$ where (l, ϕ, a, cl, l')

$\in E$. A timed automaton can progress by executing discrete or delay transitions.

A discrete transitions of the form $(l,v) \xrightarrow{a} (l',v')$ correspond to the execution of transition $l \xrightarrow{\phi,a,u} l'$ which is enabled if the clock constraint ϕ is satisfied by the clock valuation v and invariant of target location $\mathbb{I}_{l'}$ is satisfied by v'. The clock valuation v' is obtained by applying clock update u on v:

$$\frac{(l \xrightarrow{\phi,a,cl} l') \wedge \phi(v) \wedge \mathbb{I}_{l'}(v'), v' = u(v)}{(l,v) \xrightarrow{a} (l',v')} \tag{1}$$

A timed transitions are of the form $(l,v) \xrightarrow{d} (l', v+d)$, where the clock valuation v is incremented by some delay $d \in \mathbb{T}$, i.e. $v+d$ and there exists no enabled by guard edge $(l,l^{'},\phi,cl,d,a)$. It can be *delayable* if there exists $0 \leq d_1 < d_2 \leq d$ such that $v+d_1 \vDash \phi$ and $v+d_2 \nvDash \phi$ or *eager* if $v \nvDash \phi$ whereas *lazy* denotes an edge which is neither delayable, nor eager, they cannot block the time progress. Therefore, automaton can wait in a location and letting time pass as long as the invariant $\mathbb{I}_l$ for that location remains true.

$$\frac{\forall d \leq d' \leq d.\mathbb{I}_l(d')}{(l,v) \xrightarrow{d} (l, v+d)} \tag{2}$$

Time sequences. Given a finite set of actions Σ, the set $(\Sigma \cup \mathbb{R}_{\geq 0})^*$ of all finite-length real-time sequences over Σ will be denoted TS(Σ). A empty sequence is denoted by $\varepsilon \in$ TS(Σ). Given $\Sigma^{'} \subset \Sigma$ and $\rho \in$ TS(Σ), $P_{\Sigma'}(\rho)$ denotes the projection of ρ to $(\Sigma^{'} \cup \mathbb{R}_{\geq 0})^*$, obtained by *erasing* from ρ all the actions not in $(\Sigma^{'} \cup \mathbb{R}_{\geq 0})^*$. Similarly, $DP_{\Sigma'}(\rho)$ denotes the (discrete) projection of ρ to $\Sigma^{'}$. For example, if Σ = $\{x,y\}$, $\Sigma^{'}$ = $\{x\}$ and ρ = $x1y2x3$, then $P_{\Sigma'}(\rho)$ = $x3x3$ and $DP_{\Sigma'}(\rho)$ = xx. The time spent in a sequence ρ, denoted time(ρ) is the sum of all delays in ρ , for example, time(ε) = 0 and time(x1y0.7) = 1.7. According to TLTS, a state $s \in S_{\mathtt{A}}$ is reachable if there exists $\rho \in$ TS(Σ) such that $s_0^{\mathtt{A}} \xrightarrow{\rho} s$. The set of reachable states of A is denoted by RS(A).

Definition 2.3. *(Timed Input/Output Automata)*. A timed input/output automaton (TIOA) $(L, l_0, \mathbb{C}, \Sigma_i, \Sigma_o, \mathbb{I}, E)$ is a timed automata where the set of actions is classified into three disjoint sets, i.e. $\Sigma_\tau = \Sigma_i \cup \Sigma_o \cup \{\tau\}$, where Σ_i denotes the set of input actions; Σ_o set of output actions. Actions in $\Sigma_i \cup \Sigma_o$ are observable actions and TIOA is called observable if none of its edges is labeled by τ. For the formal syntax and semantics of TIOA we refer the reader to [17] and [19].

Input-enabled TIOA: A TIOA A is called input-enabled w.r.t $\Sigma^{'} \subseteq \Sigma_i$ if it can accept any input in $\Sigma^{'}$ at any state, i.e. $\forall s \in$ RS(A).

A is *deterministic*, iff $\forall l, l^{'}, l^{''} \in$ RS(A).$\forall a \in (\Sigma \cup \mathbb{R}_{\geq 0})$ (delays or actions), whenever $l \xrightarrow{a} l' \wedge l \xrightarrow{a} l''$ then l' = l'', which means for each timed word, the target location of action is always uniquely known. Actions and time in A are two important parameters that can affect the determinism of automaton. Figure 8 (a) shows the non-deterministic behavior of the TIOA. There are two distinct transitions outgoing from location l_2 that are labeled by two time words which are not disjoint, i.e at time cl == 6, there will be two possible

target locations, hence non-determinism. Similarly, in location l_8, there are two distinct transitions labeled by action $in[2]!$ which shows non-deterministic behavior of automaton.

A TIOA is *non-blocking*, if for any location l and any t such that $\forall l \in \mathrm{RS}(\mathtt{A}).\forall t \in \mathbb{R}_{\geq 0}.\exists \rho \in \mathrm{TS}(\Sigma_o \cup \{\tau\})$: time(ρ)= $t \wedge l \xrightarrow{\rho}$. The non-blocking property ensures that A will not block time in any environment. This property ensures that a system will not force or rush its environment to deliver an input, and vice versa, the environment will never force outputs from the system. Time is common for both the system and its environment, and neither controls it.

A *timed word* (σ, ρ) of length n is an element of $(\mathbb{T}^* \times \Sigma^*)$, where actions $\sigma = \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n$ are paired with *time sequence* $\rho = \rho_1 \cdot \rho_2 \cdot \ldots \cdot \rho_n$ such that ρ is strictly monotonically increasing. A timed word pair (σ, ρ) is viewed as an action σ_i taken by automata A after ρ_i time units since A is started. The elapsed time ρ_i is called a time-stamp of the action σ_i. A *timed trace* is a sequence of time stamped actions followed with a delay $TTraces(\mathtt{A}) = (\sigma_1, \rho_1) \cdot (\sigma_2, \rho_2) \cdot \ldots \cdot (\sigma_n, \rho_n)$, where $\forall 0 \leq i \leq j \leq |n|$ such that if (σ_i, ρ_i), $(\sigma_j, \rho_j) \in \mathtt{A}$, then condition $(\sigma_i, \rho_i) \leq (\sigma_j, \rho_j)$ is satisfied.

A path in A is a finite sequence of consecutive transitions:

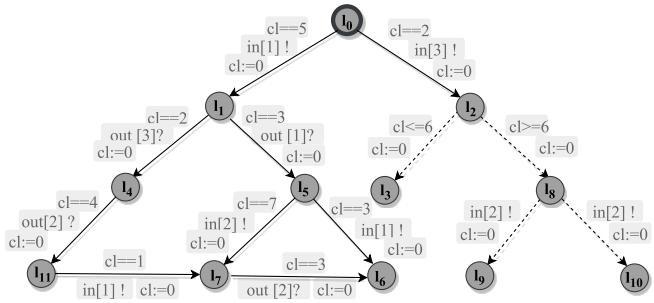
$$P = l_0 \xrightarrow{\phi_1, a_1, u_1} l_1 \xrightarrow{\phi_2, a_2, u_2} \cdots \; l_{n-1} \xrightarrow{\phi_n, a_n, u_n} \cdots \; l_n$$

where $l_{i-1} \xrightarrow{\phi_i, a_i, u_i} l_i \in E$ for every $1 \leq i \leq n$. A path is correct and accepting if it starts from an initial location and ends in a final location.

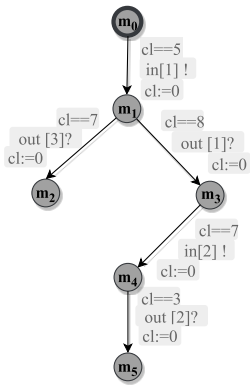
A run of A with initial location l_0 and a clock valuation v_0 is a sequence of states and transitions $(l_0, v_0) \xrightarrow{d_1, a_1} (l_1, v_1) \xrightarrow{d_2, a_2} \cdots (l_{n-1}, v_{n-1}) \xrightarrow{d_n, a_n} (l_n, v_n)$ where $v_0 \vDash \mathbb{I}(l_0)$, and for each 1 ≤ i ≤ n $(l_{n-1}, v_{n-1}) \xrightarrow{d_1, a_1} (l_n, v_n)$ is associated with the transition $(l_{n-1}, v_{n-1}) \xrightarrow{\phi_i, a_i, u_i} (l_n, v_n)$. We consider that the behavior of real-time systems can be described using timed traces, where each delay refers to the time elapsed since the automata A started.

The set of timed traces of TIOA is $TTraces(\mathtt{A}) = \{\, \rho \mid \rho \in \mathrm{TS}(\Sigma_\tau) \wedge l_0^{\mathtt{A}} \xrightarrow{\rho} \}$. The set of observable timed traces of A is defined to be $ObsTTraces(\mathtt{A}) = \{\, P_\Sigma(\rho) \mid \rho \in \mathrm{TS}(\Sigma_\tau) \wedge l_0^{\mathtt{A}} \xrightarrow{\rho} \}$.

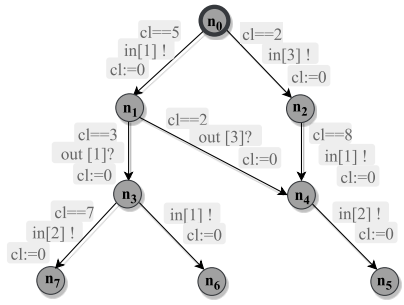
Example 2.1. Consider the timed I/O automata specification Spec shown in Figure 8 (a) where $in[1]!$, $in[2]!$, $in[3]!$ denote the inputs to the system and $out[1]?$, $out[2]?$, $out[3]?$ denote the outputs produced in response to those inputs. The Spec can be expressed in natural language as follow: exactly at 5 time units after start the system receives the input $in[1]!$ and produces either output $out[3]?$ exactly at 2 time units or, failing to do that, produces output $out[1]?$ exactly at 3 time units. The clock cl is set to 0 just after passing each transition. A run of Spec is R = $(l_0, 0) \xrightarrow{0.2} (l_0, 0.2) \xrightarrow{3.8} (l_0, 4) \xrightarrow{1} (l_0, 5) \xrightarrow{in[1]!} (l_1, 0) \xrightarrow{2.5} (l_1, 2.5) \xrightarrow{0.5} (l_1, 3) \xrightarrow{out[1]?} (l_5, 0) \xrightarrow{3.5} (l_5, 3.5) \xrightarrow{3.5} (l_5, 7) \xrightarrow{in[2]!} (l_7, 0) \xrightarrow{2} (l_7, 2) \xrightarrow{1} (l_7, 3) \xrightarrow{out[2]?} (l_6, 0)$, the timed trace associated with R is Seq(R) = $0.2 \cdot 3.8 \cdot 1 \cdot in[1]! \cdot 2.5 \cdot 0.5 \cdot out[1]? \cdot 3.5 \cdot 3.5 \cdot in[2]! \cdot 2 \cdot 1 \cdot out[2]?$. The associated time trace is Traces(Seq(R)) = $(5 \cdot in[1]!) \cdot (8 \cdot out[1]?) \cdot (15 \cdot in[2]!) \cdot (18 \cdot out[2]?) \cdot 0$. From statement (3), (4), we have Spec After $(5 \cdot in[1]!) \cdot 0 = \{(l_1, 0)\}$, Spec After $(5 \cdot in[1]!) \cdot (8 \cdot out[1]?).0 = \{(l_5, 0)\}$ $Out($



(a) Specification



(b) Implementation$_1$



(c) Implementation$_2$

Figure 8 – Models of TIOA specification and implementations

Spec After $(5 \cdot in[1]!) \cdot (7 \cdot out[1]?).0)$ = $\mathbb{T}$, $Out($ Spec After $(5 \cdot in[1]!) \cdot (8 \cdot out[1]?).15)$ = $\{in[2]!\} \cup \mathbb{T}$.

Parallel composition of TIOA. Given a specification TIOA model Spec = $(L, l_0, \mathbb{C}_1, \Sigma_i, \Sigma_o, E)$ and implementation model Impl = $(Q, q_0, \mathbb{C}_2, \Sigma_o, \Sigma_i, E)$ where the set of input (output) actions in Impl and output (input) actions in Spec are denoted with identical names but with different suffices ("!" and "?"). TIOA define three kinds of actions: the emission of an input *i*, the reception of an output *o*, and the occurrence of an internal action. Inputs and outputs are observable, whereas internal actions are unobservable. The emission of an action (act) is denoted by $act!$ and its co-action i.e. reception is denoted by $act?$. Thus, the input *i!* send (emission) by Spec is received (reception) by Impl as its co-action *i?*. Similarly, output *o!* send (emission) by Impl is received (reception) by Spec as its co-action *o?*. The parallel composition of two TAIO Spec and Impl is denoted by Spec $\|$ Impl = $(L \times Q, (l_0, q_0), \mathbb{C}_1 \cup \mathbb{C}_2, \Sigma_i, \Sigma_o, E)$ where transition relation E is defined as

$$\frac{l \xrightarrow{a} l' \quad q \xrightarrow{a} q'}{(l,q) \xrightarrow{a} (l',q')} \qquad \frac{l \xrightarrow{\tau} l'}{(l,q) \xrightarrow{\tau} (l',q)} \qquad \frac{q \xrightarrow{\tau} q'}{(l,q) \xrightarrow{\tau} (l,q')} \qquad \frac{l \xrightarrow{d} l' \quad q \xrightarrow{d} q'}{(l,q) \xrightarrow{d} (l',q')} \tag{3}$$

Issues with parallel composition for real-time systems. The parallel composition of two TAIO assumes that the communications between the tester (which runs the specification model) and the SUT (which is represented by the Implementation) are synchronous meaning that the tester blocks upon transmitting an input to (receiving an output from) the implementation. It is also assumed that the SUT and the tester are located at the same site and there is no communications latency. Consequently, the time instants at which the tester sends available inputs or receives expected outputs should be exactly those described by the specification. Local testers that do not tackle with non-deterministic communication delays control the tests, i.e. each time a tester observes an output, that output depends deterministically only on its inputs. The controllability of the test is an important property giving the possibility to lead the SUT into a intended situation. In the context of testing real-time systems with traditional centralized remote testing where the SUT and the tester are not located at the same site and communications may be delayed, the synchronous communication (in its strict sense) between SUT and *tester* model cannot be implemented due to the practical communication delays/latency. Therefore, communication latency is an essential challenge of remote testing that may lead to unintended interleaving of inputs and outputs. This affects the generation of inputs for the SUT and the observation of outputs that may trigger a wrong test verdict. We discuss the limitation of synchronous testing and testing with asynchronous semantics to overcome them in section 2.3.3.

2.3.2 Uppaal Timed Automata

In this work, we use UPPAAL Timed Automata [18] that extend the modeling power of TIOA with additional data types and several usability features. UPPAAL TA are based on the definition of timed automata, which is introduced in [48]. A real-time system consist of multiple, possibly concurrent components and for modelling such concurrent components, the timed automata are composed into a network of timed automata (NTA) over a common set of clocks and actions of n timed automata. Let the tuple $(L_i, l_i^0, \mathbb{C}, \Sigma, E_i, \mathbb{I}_i)$ denote an automaton $\mathtt{A}_i$ of the NTA, where, 1 $\leq$ i $\leq$ n, a location vector for the whole network $\|_i$ $\mathtt{A}_i$ is $\bar{l}$ = (l_1, $\cdots$ l_n) and the invariant of the network is a conjunction over locations vector invariants $\mathbb{I}(\bar{l}) = \bigwedge_i \mathbb{I}(\bar{l_i})$.

UTA extend the expressiveness of NTA by introducing richer variable types and a set of standard functions and predicates used in the guards and update functions of edges and in the invariants. Expressions in UTA range over clocks, booleans and integer variables and their arrays. The advantage of this extension is that the model has rich enough modelling power to represent real-time and resource constraints and at the same time to be efficiently decidable for reachability analysis. The expressions are used with the following labels:

Invariant: is an expression that refers only to clocks, constants and integer variables; it is a conjunction of conditions of the form cl < t or cl $\leq$ t where cl is a clock and t $\in \mathbb{N}$ evaluates to non-negative integer.

Guard: is an expression that evaluates to a boolean; it includes terms such as clocks, integer variables, constants and function symbols (standard Uppaal functions and those implemented in the given model); clocks and clock differences are compared only to integer expressions; guards over clocks are essentially conjunctions although expressions

over integers can be in disjunction.

Assignment: is a comma-separated list of variable updates where the expressions must only refer to clocks, boolean and integer variables and their arrays, functions of them and constants. The clocks can be updated only with integer constants.

Synchronization: Synchronous communication between concurrent components is modeled in UTA by co-use of synchronization channels and shared, bounded data variables. The label of channel on synchronized edges has either of the form *Expression*! or *Expression*? or an empty label. The expression must be side-effect free, evaluate to a channel, or channel arrays. Guard conditions declare that an edge may only be fired if a matching synchronizing second edge is enabled for firing. Time transitions are only possible in locations and are not represented by edges in the UTA graph. This is why locations are annotated with invariants. A state of the system may only stay in a certain location while the clock invariant is satisfied.

The UTA models are defined as a closed network of extended timed automata that are called *processes*. The processes are combined into a single system by the parallel composition. An example of a system composed of two automata instances referred as processes is given in Figure 9 ($Process_A$ on left and $Process_B$ on right, respectively). The *state* of

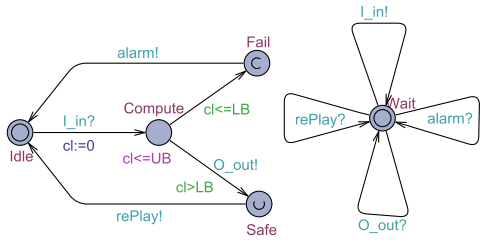


Figure 9 – A synchronous composition of two Uppaal automata

an automaton consists of its current locations and assignments to all variables, including clocks. The initial locations of the automata (of the parallel composition) are graphically denoted by an additional circle inside the location. The automata in Figure 9 have channels `I_in!`, `O_out?`, `rePlay?` and `alarm?`, where `I_in`, `O_out`, `rePlay` and `alarm` are the names of the channels. In Figure 9, $Process_B$, initially at location *Wait*, initiates the call by executing the send action `I_in!` that is synchronized with the receive action `I_in?` in $Process_A$, that is initially at location *Idle*. The location *Compute* denotes the situation where $Process_A$ computes the output.

The duration of the execution of the result is specified by the interval $[0, UB]$ which is split by guards of outgoing edges into two non-intersecting sub-intervals $[0, LB]$ and $(LB, UB]$, where the upper bound UB is given by the *invariant* `cl<=UB`, and the lower bound LB by the condition `cl>LB` of the transition $Compute \rightarrow Safe$. The model generates time words which are disjoint, i.e if `cl<=LB` the control passes to committed location *Fail* from which the control returns back immediately to initial location *Idle* by synchronizing with the receive action `alarm?` in $Process_B$. Committed locations are indicated by a location with an encircled "C". A committed location must be left immediately by executing the next transition taken in the system. If `cl>LB` the control passes to urgent location

Safe and returns to *Idle* by synchronizing with the receive action `rePlay?` in $Process_B$. An urgent location (encircled "U") must be left without letting time pass, but allows interleaving with enabled transitions from urgent locations of other automata.

The *assignment* `cl=0` on the transition ($Idle \xrightarrow{cl:=0} Compute$) ensures that the clock cl is reset when the control reaches the location $Compute$. The internal action is indicated by an absent action-label. In the following, the UTA notions described in examples above will be defined formally. For the full definition of formal syntax and semantics of UTA we refer the reader to [19] and [17].

Definition 2.4. An UTA model $\mathbb{M}$ is a tuple $(\overrightarrow{A}, Vars, Clocks, Chan, Type)$ where:

- $\overrightarrow{A}$ is a vector of processes $A_1, A_2, \cdots A_n$ (when the automaton template of a process needs to be made explicit we use double indexes, e.g. a process A^i_j denotes j-th instantiation of an automaton template A^i; for better readability we ignore the indexes of templates when it is clear from the context and the elements of a process A_j are referred to by applying the index of the process, e.g. L_j,l^0_j, T_j.
- *Vars* is a set of variables (except clocks) defined in the model. It is a union over $Vars_j$ of processes and global variables of the model.
- *Clocks* is a set of clocks such that Clocks ∩ Vars = ∅. Like *Vars*, *Clocks* is the union of all $Clocks_j$ in the processes and global clocks of NTA.
- *Type* is a type function mapping locations to types. The location types are 'committed', 'urgent' and 'normal' (their semantics is defined in [18]).

Definition 2.5. *(Configuration of UTA)* Configuration of an UTA model $(\overrightarrow{A}, Vars, Clocks, Chan, Type)$ is a triple $(\overrightarrow{l}, e, v)$ where $\overrightarrow{l}$ is a vector of locations, *e* is the valuation function of discrete variables and *v* is a clock valuation:

- $\overrightarrow{l}$ = $(l_1, l_2, \cdots l_n)$ where $l_i \in L_i$ is current location of process A^i;
- *Vars* $\rightarrow \prod_i dom(v_i)$ maps every variable $v_i \in$ *Vars* to its value;
- *Clocks* $\rightarrow \mathbb{R} \geq 0$ maps the clocks to non-negative real numbers.

Configuration of the model corresponds to the state in the definition of NTA. The vector $\bar{l}$ is called situation (currently occupied locations in the automata of the model), pair $(\bar{l}, e)$ denotes the discrete part and *v* the continuous part of the configuration.

UTA state like TA state evolves either through enabled actions or delays. These steps define the behavior of the model. For configuration $(\overrightarrow{l}, e, v)$ a local action is enabled if there is an enabled internal transition in the underlying NTA. A synchronized action step is enabled iff for a channel *b* there exists synchronizable by channel *b* transition in the underlying NTA. A delay step with delay *d* is enabled iff such delay step is allowed in the underlying NTA.

Definition 2.6. Let $\mathbb{M}$= $(\overrightarrow{A}, Vars, Clocks, Chan, Type)$ be a UTA model. A sequence of configurations:

$\langle(\overrightarrow{l}, e, v)\rangle^K$ = $\langle(\overrightarrow{l}, e, v)^0, (\overrightarrow{l}, e, v)^1, \cdots\rangle$ of length $K \in \mathbb{N} \cup \infty$ is called a well-formed sequence for $\mathbb{M}$ iff

- $(\overrightarrow{l}, e, v)^0$ = $(l_1^0, \cdots l_n^0), [Vars \mapsto (0)^{|Vars|}], [Clocks \mapsto (0)^{|Clocks|}])$
- if $K < \infty$ then for $\langle(\overrightarrow{l}, e, v)\rangle^K$ no further steps is enabled
- if $K = \infty$ and $\langle(\overrightarrow{l}, e, v)\rangle^K$ contains finitely many k such that $(\overrightarrow{l}^k, e^k) \neq (\overrightarrow{l}^{k+1}, e^{k+1})$, then eventually every clock exceeds every bound $(\forall x \in Clocks, \forall c \in \mathbb{N}, \exists k : v^k(x) > c)$

Definition 2.7. *(Timed trace of UTA)*. A well-formed sequence of $\mathbb{M}$ is a timed trace if for every $k < K$, the two subsequent configurations k and $k + 1$ are connected via a simple action step, a synchronized action step, or a delay step, i.e. $(\overrightarrow{l}^k, e^k) \xrightarrow{a} (\overrightarrow{l}^{k+1}, e^{k+1})$ or $(\overrightarrow{l}^k, e^k) \xrightarrow{d} (\overrightarrow{l}^{k+1}, e^{k+1})$ or $(\overrightarrow{l}^k, e^k) \xrightarrow{\tau} (\overrightarrow{l}^{k+1}, e^{k+1})$. Let $\mathbb{M}$ be a UTA model, then the trace semantics of $\mathbb{M}$, denoted TTraces(M), is the set of well-formed traces.

n-Ports UTA, we represent a multi-ports TIOA in UTA by splitting the transition with multiple simultaneous synchronization actions to a sequence of transitions each labeled with exactly one I/O-action and connected via committed locations, so that all ports of such group are updated instantaneously in the order they are specified in the tuple. In Figure 10, the labels on the i/o actions (encoded as elements of channel arrays 'in' and 'out') represent the transitions and the transition tuple (l_0, l', $in[1]!$ /($out[1]?$, $out[3]?$)) is represented by sequence of transitions each labeled with exactly one action and connected via committed locations, l_0 represents the *idle*, and l' represents the *Done* location. Let P_{l_n} denotes a set of ports accessible in some geographic location l_n where $n \in \mathbb{N}$; I is

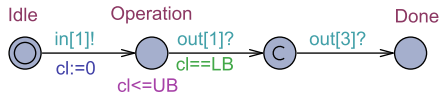


Figure 10 – Modelling pattern of multiport timed automata

a n-tuple (I_1, I_2,...I_n), where I_i is finite set of inputs at port i, $I_i \cap I_j = \phi$ for $i \neq j$ and i, j = 1,...$n \in \mathbb{N}$. Similarly, O is a n-tuple (O_1, O_2,...O_n), where O_i is finite set of outputs at port i, $O_i \cap O_j = \phi$ for $i \neq j$ and i, j = 1,...$n \in \mathbb{N}$. Each port may receive outputs of other port, i.e O = $(O_1 \cup \{\varepsilon\}) \times (O_2 \cup \{\varepsilon\}) \times ... \times (O_n \cup \{\varepsilon\})$, here $\{\varepsilon\}$ denotes the empty output in response to input to SUT.

2.3.3 Timed Input/Output Conformance Relation

The MBT theory makes an assumption that system under test can be represented by formal model that is system representation on appropriate level of abstraction. This assumption is referred to as a test hypothesis. Given two models - the specification Spec and the system model Impl, MBT theory studies the problem of how these two are relate to each-other. Specifically, the testing methods applied in the thesis rely on the timed input-output conformance (tioco) relation, an extension of ioco [46].

IOCO theory reasons about black-box conformance testing. The main difference is that ioco uses the notion of quiescence, according to which the absence of outputs is observable. In tioco we do not use quiescence because we want timeouts to be explicitly specified. For instance, we do not allow specifications stating "an event must eventually occur" but "an event must occur within x time units". Apart from this important difference, tioco is similar in spirit to ioco: intuitively `Impl` conforms to `Spec` if for each observable behavior specified in `Spec`, the possible outputs of `Impl` after this behavior is a subset of the possible outputs of `Spec`. In tioco time delays are included in the set of observable outputs. This permits to capture the fact that an implementation producing an output too early or too late (or never, whereas it should) is non-conforming.

From *definition 2.3* of TIOA, we assume that the specification `Spec` of the system to be tested is given as a non-blocking TAIO and the `SUT` `Impl` can be modeled as an input-complete, and non-blocking. In order to define the conformance relation, we recall the timed input/output conformance relation (`tioco`) introduced by [13, 14, 21]. They propose extension of `ioco` relation with timing constraints including clock valuations with the set of observable actions. Given TIOA `A` and timed traces $TTraces(A)$, `A` After ρ is the set of all location that can be reached after timed sequence ρ, the `tioco` relation is defined as:

$$\texttt{Impl}\ \textbf{tioco}\ \texttt{Spec} \equiv \forall \rho \in \text{TTraces(}\ \texttt{Spec}\text{)} : \mathit{Out}(\texttt{Impl}\ \text{After}\ \rho) \subseteq \mathit{Out}(\texttt{Spec}\ \text{After}\ \rho)$$

During testing, tester emulates the specification either by letting time elapsing or by applying input to the `SUT` and waiting for the expected outputs from SUT. Upon reception of expected outputs, the tester emits a test verdict (pass, fail, inconclusive). The verdict shows correctness in the sense that `SUT` conforms to the specification if the expected outputs (specified and correct time instants) exactly those described by the specification are observed. Due to inherent non-determinism of distributed systems the natural choice is online testing where the tester model is executed in lock step with the SUT. The communication between the tester and the `SUT` involves controllable inputs of the `SUT` and observable outputs of the `SUT` which makes possible to detect tioco violations.

Example 2.2. Consider the timed I/O automata specification `Spec` and implementations $\texttt{Impl}_1$, $\texttt{Impl}_2$ shown in Figure 8. Based on `tioco` relation, we can verify that if $\texttt{Impl}_1$ and $\texttt{Impl}_2$ conform to `Spec` as shown in Table 4, 2.

Table 1 – $\texttt{Impl}_1$ `tioco` `Spec`

$\mathit{Out}(\ \texttt{Spec}\ \text{After}\ (5 \cdot in[1]!)) =$	$\mathbb{T}$
$\mathit{Out}(\texttt{Impl}_1\ \text{After}\ (5 \cdot in[1]!)) =$	$\mathbb{T}$
$\mathit{Out}(\ \texttt{Spec}\ \text{After}\ (5 \cdot in[1]!) \cdot 8) =$	$\{out[1]?\} \cup \mathbb{T}$
$\mathit{Out}(\texttt{Impl}_1\ \text{After}\ (5 \cdot in[1]!) \cdot 8) =$	$\{out[1]?\} \cup \mathbb{T}$
$\mathit{Out}(\ \texttt{Spec}\ \text{After}\ (5 \cdot in[1]!) \cdot 7) =$	$\{out[3]?\} \cup \mathbb{T}$
$\mathit{Out}(\texttt{Impl}_1\ \text{After}\ (5 \cdot in[1]!) \cdot 7) =$	$\{out[3]?\} \cup \mathbb{T}$

Table 2 – $\texttt{Impl}_2$ ~~`tioco`~~ `Spec`

$\mathit{Out}(\ \texttt{Spec}\ \text{After}\ (5 \cdot in[1]!) \cdot (7 \cdot out[3]?)) =$	$\mathbb{T}$
$\mathit{Out}(\texttt{Impl}_2\ \text{After}\ (5 \cdot in[1]!) \cdot (7 \cdot out[3]?)) =$	$\{in[2]!\} \cup \mathbb{T}$
$\mathit{Out}(\ \texttt{Spec}\ \text{After}\ (5 \cdot in[1]!) \cdot (8 \cdot out[1]?).18) =$	$\{out[2]?\} \cup \mathbb{T}$
$\mathit{Out}(\texttt{Impl}_2\ \text{After}\ (5 \cdot in[1]!) \cdot (8 \cdot out[1]?).18) =$	$\breve{}$

Synchronous Testing: According to UTA channels semantics the synchronization by using channels cannot be fully implemented due to realistic communication delays. Only when assuming the SUT and tester are located at the same site and there is no communications latency the synchrony assumption holds. In the context of testing real-time systems with centralized remote testing where the SUT and the tester are not located at the same site and communications may be delayed, the synchronous communication between SUT and *tester* model cannot be implemented in rigorous sense due to the communication delays.

To address the problem, [38] proposed an asynchronous semantics with explicit communication delays. They proposed Δ-testability criterion (where Δ describes the communication latency) with two additional assumptions about the communication in TAIO parallel composition model. The model is centered around a $2FIFO(\bowtie, \Delta)$ architecture that consists of:

1. One first-in-first-out ($FIFO$) queue for each direction of the communication between the SUT and the tester.

2. A communication latency bounded by Δ. The symbol $\bowtie$ stands for either $\leq$ or $=$. The tioco relation for asynchronous parallel composition $\|_{async}$ is defined in terms of asynchronous trace semantics [38] as in Definition 2.8.

Definition 2.8. *(Asynchronous semantics for TIOA).* Let $\mathtt{A} = (L, l_0, \mathbb{C}, \Sigma_i, \Sigma_o, \mathbb{I}, E)$ be a TIOA with no silent actions. Let $\bowtie \in \{\leq, =\}$ and $\Delta \in \mathbb{N}$. The asynchronous semantics for A that is an IOTTS $(\Sigma_i, \Sigma_o, \Lambda_{\Sigma_i \cup \Sigma_o})$ is defined as follows

$$\langle |\mathtt{A}| \rangle^{\bowtie\Delta} = \langle (L \times \mathbb{R}^{\mathbb{C}}_{\geq 0}) \times (\mathbb{R}_{\geq 0} \times (\Sigma_i \cup \Sigma_o))^* \times (\mathbb{R}_{\geq 0} \times (\Sigma_i \cup \Sigma_o))^*, (l_{0,0}), (\Sigma_i, \Sigma_o, \Lambda_{\Sigma_i \cup \Sigma_o}), M_{\bowtie\Delta} \rangle$$

where $\Lambda_{\Sigma_i \cup \Sigma_o} = \tau_a \mid a \in \Sigma_i \cup \Sigma_o$ is the set of silent actions. An asynchronous state is of the form $((l,v),p,q)$ where p and q are input and output queues respectively. The set of asynchronous moves, $M_{\bowtie\Delta}$ is defined by the following five rules:

$$\frac{}{((l,v),p,q) \xrightarrow{a?} ((l,v),p.(0.a?),q)} \quad \text{(r1)}$$

$$\frac{}{((l,v),(\delta.a?).p,q) \xrightarrow{\tau_a?} ((l',v[\mathtt{C}:=0]),p,q)} \quad \text{(r2)}$$
$$l \xrightarrow{\phi,a?,\mathtt{C}} l' \wedge v \models \phi \wedge \delta \bowtie \Delta$$

$$\frac{}{((l,v),p,q) \xrightarrow{t} ((l,v+t),p+t,q+t)} \quad \text{(r3)}$$

$$\frac{}{((l,v),p,(\delta.b!).q) \xrightarrow{\tau_b!} ((l,v),p,q)} \quad \text{(r4)}$$
$$\delta \bowtie \Delta$$

$$\frac{}{((l,v),p,q) \xrightarrow{\tau_b?} ((l',v[\mathtt{C}:=0]),p,q(0.b!))} \quad \text{(r5)}$$
$$l \xrightarrow{\phi,b!,\mathtt{C}} l' \wedge v \models \phi$$

The rules *r1* and *r2* (resp *r5* and *r4*) are dual and they correspond to the transmission and the reception of an input (resp. output). The rule *r3* corresponds to the time elapsing. The time elapsing operation on a queue is defined by $((\delta \cdot a).p)+t = (\delta+t, a).(p+t)$. Notice

that $\langle|\mathtt{A}|\rangle^{\bowtie\Delta}$ is input-complete because the transmissions of the inputs do not require to check the clock constraints. The receptions of the pending inputs require to check for the validity of clock constraints. The length of each queue is unbounded. Based on $\langle|\mathtt{A}|\rangle^{\bowtie\Delta}$, we can define asynchronous runs, asynchronous execution sequences and asynchronous timed traces in $ATTraces_{\bowtie\Delta}(\mathtt{A})$ = TTraces($\langle|\mathtt{A}|\rangle^{\bowtie\Delta}$). Asynchronous timed traces are remote observations of local timed traces at communicating automata. The execution order of actions may differ from the observation order: this happens when inputs and outputs interleave in the communication channels. We characterize remote observations that may lead to action interleaving by introducing the notion of Δ-testability [38].

Definition 2.9. *(Δ-testability).* Let $\mathtt{A} \in$ TIOA and $TTraces(\mathtt{A}) = (t_i \cdot a_i)_{i=1..n} \cdot t_{n+1}$. The timed traces in $TTraces(\mathtt{A})$ are Δ-*testable* if,

- either n = 0,
- or $(t_i \cdot a_i)_{i=1..n-1}$ is Δ-*testable* and $a_n \in \Sigma_o$,
- or $(t_i \cdot a_i)_{i=1..n-1}$ is Δ-*testable* and if $a_n \in \Sigma_i$, then for every $t_b \in \mathbb{R}_{\geq 0}$, every $b \in \Sigma_o$, and every $k \in [1..n-1]$ such that $b! \in$ **out** $([\mathbb{A}]$ **after** $\rho[1..k] \cdot t_b)$, it holds that $t_n - t_b > 2\Delta$.

A is Δ-*testable* if every trace in $TTraces(\mathtt{A})$ is Δ-*testable*.

Proposition 2.1. *Let* A *be a* TIOA *and* l, $\rho \in TS(\langle|\mathtt{A}|\rangle^{\bowtie\Delta})$ *such that* $l^0 \xrightarrow{\rho} l$. A *is* Δ-*testable iff* $p(s)$ *is non empty implies* $q(s)$ *is empty.*

According to Proposition 2.1, Δ-*testability* implies that at most one queue is non empty at every reachable state. However, Δ-*testability* does not guarantee that the sizes of the queues are bounded. A fast environment can increase the size of the input queue by sending repetitively the inputs faster than the latency. It is shown in [38] under what conditions Δ-*testability* specifications are controllable. Indeed, having the condition that output response to each input stimulus arrives before the next input is given the outputs transmitted earlier are received before the transmission of new inputs. Thus, each observed output depends on input transmitted earlier and the under this assumption test is controllable. Given the maximum signal propagation delay is Δ the delay between the two consecutive test inputs must be strictly greater than 2Δ. In brief, Δ-*testability* criterion takes advantage of the timing information that is not available in untimed models. Another important corollary of Δ-*testability* criterion is that if the specification is Δ-*testable* then, the asynchronous execution of the synthesized test cases is as simple as the synchronous execution, the tioco conformance is preserved and the tester can control the test. This result provides one of main motivations of current thesis to implement the distributed test execution environment that can implement the criterion of Δ-*testability* instead of 2 Δ testability.

2.4 Chapter Summary

In this chapter, we have discussed the basics of online model-based testing and discussed the advantages compared to manual/offline black-box testing. We have also discussed the MBT taxonomy step by step including: *building the* SUT *model and its environment*, *verification of models*, *generating abstract tests and concretizing them into executable* on SUT tests and analyzing test results. We also discussed the semantic foundation for modelling real-time systems with timed automata. We showed how the specifications

are transformed from timed automata to more expressive formalism Uppaal timed automata (UTA) to exploit the power of Uppaal simulation, verification and testing tools. We introduced the *n-ports* timed automata and showed how a multiple-port distributed real-time system test models can be build with UPPAAL modelling formalism by considering timing-constraints such as urgent, committed states, clock invariant etc. Finally, we introduced the concept of timed input/output conformance relation which is used as test pass criteria confirming the conformance of SUT to its specifications.

3 Distributed Testing

3.1 Chapter Overview

In this chapter, we present the main results of thesis. We present the test architecture for testing distributed systems and discuss its components such as adapter, coordination between adapter and local testers. We discuss several hypothesis and definitions required to implement our approach. We also discuss in detail the issues in distributed testing and elaborate them with examples. We introduce the centralized tester partitioning algorithms and compare their performances. We propose constructive algorithms to generate a set of synchronizing local testers and demonstrate how distributed synchronizing tester models are deployed across geographical locations. We present test scenario where a centralized testing cannot be applied. We also demonstrate how the presented approach is capable of addressing controllability and observability issues and discuss the limits of proposed algorithms.

3.2 Communication and Coordination Hypothesis of Distributed Testing

In this chapter, model-based testing of distributed real-time systems is considered under the assumptions that test and SUT interact via physically distributed interfaces and there is possibility to synchronize the local testers to coordinate test activities. We consider a distributed SUT which consists of several geographically distributed locations and provides access to any of its ports needed for test input/output communication. An example of local test configuration is depicted in Figure 11. A local tester at each location communicates with SUT local port via (customized for the tester) adapter.

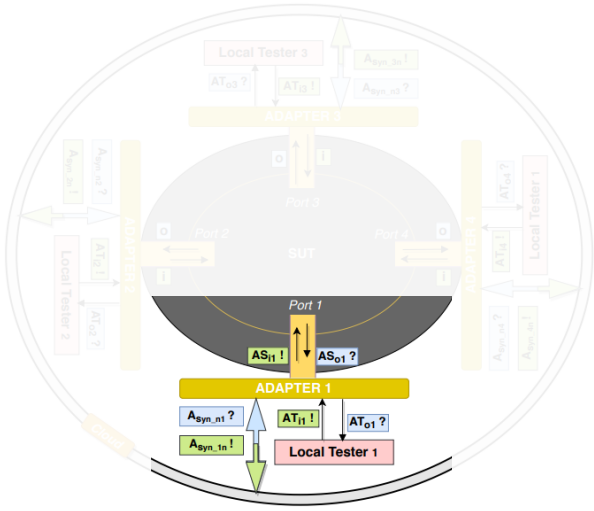


Figure 11 – New Distributed Test Architecture

Assume the SUT model has an input actions alphabet I, output actions alphabet O and internal actions. Since internal actions are not observable at ports we defer them unless they influence observable behavior at ports. Sets I and O are mapped to the set P of ports, p: $\mathtt{I} \cup \mathtt{O} \mapsto \mathtt{P}$, which satisfies following conditions:

- $\mathsf{I} \cap \mathsf{O} = \emptyset$
- p is surjective and
- $\mathtt{p}(act_i) = \mathtt{p}(act_j)$ implies $act_i = act_j$, where $act_i , act_j \in \mathtt{I} \cup \mathtt{O}$.

The emission of an action (act $\in$ I or act $\in$ O) is denoted by $act!$ and its co-action i.e. reception is denoted by $act?$. Let O = (o^1, o^2, ·, o^n) be a n-tuple, where o^k is an output at port *k*. A transition triggered by a single input can lead to outputs at more than one port in any order, i.e (o^1, ..., o^n) $\in$ O. The input *i!* sent by tester is received as a reception action by SUT (denoted by *i?*). Similarly, output *o!* send (emission) by SUT is received (reception) by tester as its co-action *o?*. For coordinated input actions at different (geographically separated) ports the local tester models synchronize actions on symbolic level using UTA channels. For instance in Figure 11, $AS_{i1}!$ and $AS_{o1}?$ denote local channels between SUT and local adapter at port 1; $AT_{i1}!$ and $AT_{o1}?$ are channels between the local adapter and local tester; and A_{Syn_1n} is the channel for coordination between the local adapter and other local adapters to synchronize the local testers where n represents the port number. It is assumed that each local adapter communicates with the SUT through its local port and with other adapters through a reliable communication medium independent of the SUT, i.e. the coordination message delivery is assumed to be reliable which means there is no message loss and propagation delay larger than Δ that may violate the test run timing. To formalize these considerations following assumptions are made:

(A.1) The MBT is interpreted in a standard way, i.e. as conformance testing that compares the expected behaviors described by the Spec model with the observed behaviors of an actual implementation (SUT). The Spec is described by Uppaal Timed Automata (UTA) which can express the non-deterministic behavior of distributed systems. Further, it is assumed that the behavior of SUT can be depicted by the Spec model and it is assumed that SUT is input-enabled but Spec does not have to be [56].

(A.2) The propagation time d which is required to exchange message between different components of distributed systems is bounded by a finite time interval [T_{lb}, T_{ub}], where lb and ub denote lower and upper bound respectively. In addition, $\mathtt{d}_{tt}$ represents the propagation time between local testers and $\mathtt{d}_{st}$ between a local tester and the SUT in case it is not negligible.

(A.3) The SUT is reactive and cannot produce outputs autonomously, which means outputs are produced only in response to reception of input as specified in Spec. SUT reacts to input by producing at most one output at each port within certain time bound.

(A.4) Both centralized and distributed tester models are deterministic. The SUT models in both cases are assumed to be equivalent and if they include control flow non-determinism then corresponding non-deterministic edges in both SUT models are mutually synchronized to avoid diverging reactions in the SUT models.

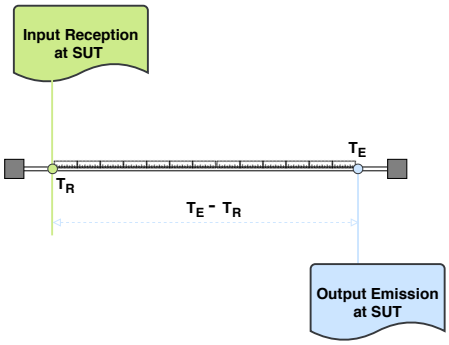


Figure 12 – Computation Time of SUT

Property 3.1. *From the assumptions* A.1 – A.3, *we conclude that the computation time* C_{SUT} *of* SUT *is bounded by finite values where* C_{SUT} *represents the time interval from the moment when an input is received (*T_R*: reception of input) by the* SUT*, until its processing terminates and the output is produced at* T_E*. Thus,* SUT *response is assumed to be bounded by* C_{SUT} *i.e.* $C_{SUT} \geq T_E - T_R$ *as shown in Figure 12. We assumed that* SUT *response time to given test input does not depend on whether the input is sent by centralized remote tester of by some local component of the distributed tester.*

Property 3.2.

The maximum progress of test execution means that both the centralized remote tester and distributed testers try to execute their i/o actions without exceeding their enabling time intervals. Due to the fact that the remote tester is not at the same geographical location with SUT *ports, the propagation latency may affect the arrival time of inputs to* SUT *test ports and thus trigger unintended reactions. Therefore, in the presence of propagation latency, remote tester has to be fast enough to produce inputs so that they will be received by* SUT *in an expected time window. This is not the case with distributed local testers as they are directly attached to the* SUT *local ports and their communication delay with* SUT *local component may be assumed to be negligible.*

Property 3.3.

In IOCO testing it is generally assumed that SUT *can wait for input in any state unbounded time. In real time systems input waiting time (e.g. for sensor data) is strictly bounded. From the assumptions and property 3.2, we conclude that the waiting time* W_{SUT} *of the* SUT *is bounded by finite values and the propagation time* d_{st} *between tester and* SUT *also must have upper bound. In untimed system models the* W_{SUT} *cannot be quantified, instead the abstract property quiescence is used to express bounded input receiving enabledness time of the* SUT *after it has processed its previous input and terminated.*

Property 3.4.

By assumptions each local tester receives two kinds of inputs: coordination message from other local testers and outputs from the local component of SUT. *We assume that the waiting time* $\mathtt{W}_{st}$ *of the tester for every expected output from* SUT *and waiting time* $\mathtt{W}_{tt}$ *of receiving synchronization signals from other local testers is bounded by finite values. This is essential for concluding any verdict whether* SUT *conforms to* Spec, *and it can be done only if* $\mathtt{C}_{SUT}$ *and propagation time* $\mathtt{d}_t$ *have upper bounds.*

Example 3.1.

Consider two actions act_p and act_q executed at SUT ports in location ℓoc_p and ℓoc_q respectively. Let assume action act_p has order constraint "act_p must be executed before act_q" or timing constraints relatively to action act_q. Any time the tester at ℓoc_p applies an input to SUT, it has to send a coordination message (A_{Syn_pq}) immediately to local testers at other locations ℓoc_q. Coordination message A_{Syn_pq} contains the model status information which allows tester at location ℓoc_q inject its message timely, i.e. assure the controllability and observability of actions at SUT ports.

Property 3.5.

If an action act_q *after the occurrence of input* act_p *is an output produced at* ℓoc_q *by* SUT *in response to input* act_p *applied at* ℓoc_p, *then tester at location* ℓoc_q *allows to receive* act_q *from* SUT *and synchronization actions* A_{Syn_pq} *in any order. After receiving* A_{Syn_pq} *at* ℓoc_q *local tester can verify whether timing constraints of* act_q *are respected. In this case, it is not required that the maximum propagation time* $\mathtt{d}_{tt}$ *among tester has to be smaller than the minimum computation time* $\mathtt{C}_{SUT}$ *of* SUT *but both receiving events should fit into* Δ, *and interpreted as a pair of simultaneous events. Hence, waiting by tester to receive* A_{Syn_pq} *is not blocking timing-wise.*

Property 3.6.

If an action act_q *after the occurrence of input* act_p *is an input to the* SUT, *then local tester at* ℓoc_q *has to wait for a coordination message* A_{Syn_pq} *from local tester at* ℓoc_p *which signals that tester at* ℓoc_q *can proceed with its input action* act_q.

To guarantee the order of actions and timing constraints at different local ports, it is required in general that the maximum propagation time of A_{Syn_pq} *between testers has to be smaller or equal to the minimum computation time* $\mathtt{C}_{SUT}$ *of the* SUT *to guarantee that waiting by local tester to send/receive action is not blocking. According to* Property 3.5, *for* Δ*-testability it suffices that the waiting time for* A_{Syn_pq} *does not exceed* Δ - $\mathtt{C}_{SUT}$.

Controllability. Controllability in distributed MBT is the capability of local testers to manage the test activity by sending inputs to SUT and receiving outputs from SUT in a specific order and timing specified by the test model. For instance, given a test sequence $\langle i_p/\{o_p\}\rangle \cdot \langle i_q/\{o_q\}\rangle \cdots \langle i_l/\{o_l\}\rangle$ where $l \in \mathbb{N}$, the controllability problem occurs when the testers at ports p and q cannot guarantee that SUT will receive the inputs in given order, i.e $\mathbf{i}_p$ before $\mathbf{i}_{p+1}$, $\forall n \in l$. This problems occur due to: *(i)* when port of $\mathbf{i}_n$ is different from port of $\mathbf{i}_{n+1}$ (i.e., $q \neq p$) and *(ii)* when tester at port q (ready to send i_q but does not know when to send an input) has not received any coordination information about the occurrence of input i_n or output o_n from SUT in response to previous execution $\langle i_p/\{o_p\}\rangle$ triggered from port p.

Observability. It is the property of local testers to observe the outputs of SUT and to determine the input which causes each of the outputs. For instance, given a consecutive tests for a port p: $\langle i_n/\{o_p, o_q\}\rangle \cdot \langle i_{n+1}/\{o_p\}\rangle$ or $\langle i_n/\{o_p\}\rangle \cdot \langle i_{n+1}/\{o_p, o_q\}\rangle$. Observability problem occurs when the tester at port q receives output o_p and is not able to determine whether this output is produced by SUT in response to input $\mathbf{i}_n$ or $\mathbf{i}_{n+1}$. The sequencing of two consecutive test inputs such that only one of them contains an output leads to output-shift faults.

3.3 Generating Distributed Testers

As discussed in previous chapter, the shortcoming of the centralized remote testing approach is mitigated by decomposing the monolithic remote tester into multiple local testers which can coordinate between themselves. These local testers are directly attached to the ports of the SUT. Thus, instead of bidirectional communication between a remote tester and the SUT, only unidirectional synchronization between each local tester with others is required to update the other testers about one's local observations and control actions. In this work we suggest a two step approach to distributed tester generation where at first, a centralized remote tester is generated by applying the reactive planning online-tester synthesis method of [64], and second, a set of synchronizing local testers is derived from it by decomposing the monolithic tester into a set of location specific tester instances. Each tester instance needs to know now only the occurrence of those i/o events at other ports which determine its own behavior. Possible reactions of the local tester to these events are already specified in the centralized tester model produced in step one and do not need further return communication to the event observer. An important concern of this partitioning transformation is that it should preserve the correctness if proved once about the centralized tester so that if the centralized remote tester meets 2Δ requirement then the distributed testers with same functionality meet (one) Δ-controllability requirement.

In the following, we propose partitioning algorithms with different sets of assumptions to generate distributed local testers. The algorithms assume that the first step (generating the test model for centralized remote tester) is accomplished and the tester satisfies Δ control criteria. We discuss how the partitioning algorithms and more specifically a coordination of local testers is able to avoid distributed testing issues.

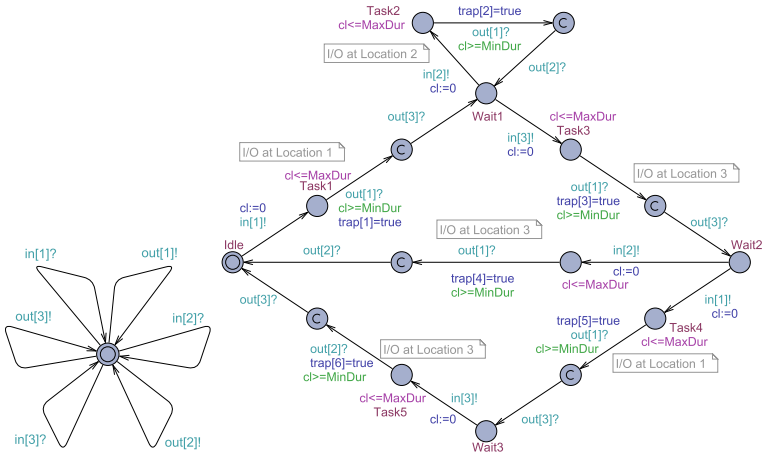


Figure 13 – SUT and Remote Tester Model

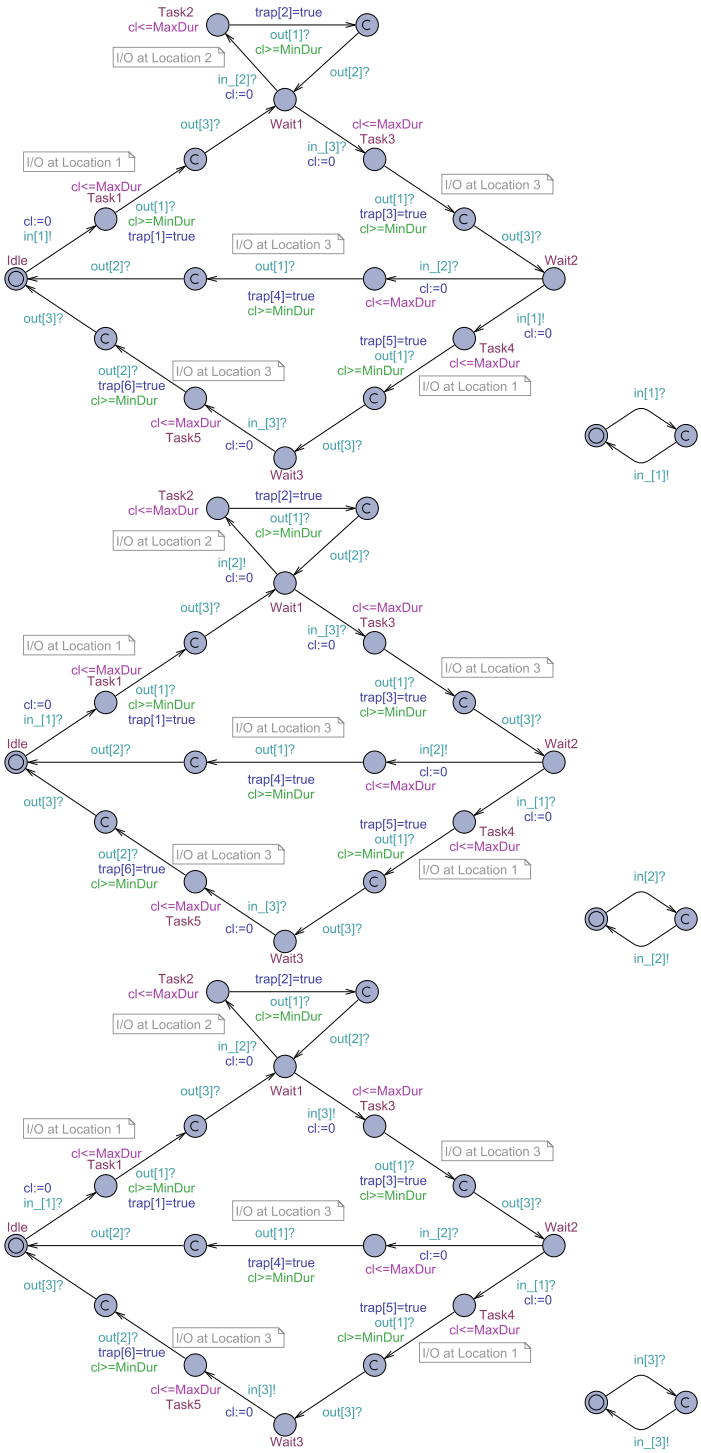


Figure 14 – Distributed Local Testers

3.4 Algorithms

Consider the remote testing architecture depicted in Figure 1 and its corresponding UTA model in Figure 13. The SUT shown in figure has 3 ports (p_1, p_2, p_3) in geographically different places with inputs/outputs $in[1]/out[1]$, $in[2]/out[2]$ and $in[3]/out[3]$ at ports $p1$, p_2 and p_3 respectively.

Let M^{RT} denote a monolithic remote tester model generated by applying the reactive planning online-tester synthesis method [64]. $Loc(\mathtt{SUT})$ denotes a set of geographically different port locations of SUT. The number of locations ranges from 1 to n, where $n \in \mathbb{N}$ i.e. $Loc(\mathtt{SUT}) = \{l_n | n \in \mathbb{N}\}$. Let P_n denotes a set of ports accessible in the location l_n.

Algorithm 1 Automated Construction of Adapter and Local Testers

input: M^{RT};
output: $||_n M^{\mathtt{DT}}\ n \in \mathbb{N}$;

1: For each l, $l \in Loc(\mathtt{SUT})$ we copy M^{RT} to M^l to be transformed to a location specific local tester instance.
2: For each M^l we go through all the edges in M^l. If the edge has a synchronizing channel and the channel does not belong to the the set of port P_l, we do the following:
   - if the channel's action is $send$, we replace it with the co-action $receive$.
   - if the channel's action is $receive$, we do nothing.
3: For each M^l we add an adapter automaton (an example is shown in Figure 14) that duplicates the input signals from M^l to SUT, attached to the set of ports P_{l_n} and broadcasts the duplicates through adapter to other local testers to synchronize the test runs at their local ports.

3.4.1 Algorithm 1

`Algorithm 1` proposed in [6] transforms the centralized testing architecture to a set of communicating distributed local testers. The resulting architecture mitigates the timing issue by replacing the bidirectional communication with a unidirectional broadcast of the SUT local i/o actions to local testers at other ports. Self-explanatory conceptual description of the algorithm is depicted in `Algorithm 1`. An example of how generated from centralized tester model of Figure 13 local tester models look like is shown in Figure 14.

Correctness of Tester Distribution Algorithm: To verify the correctness of distributed tester generation algorithm we check the bisimulation equivalence relation between the model of monolithic centralized tester and that of distributed tester. For that the models are composed by parallel compositions so that one has a role of timed words generator on i/o alphabet and other the role of timed words acceptor machine. If the i/o language acceptance is established in one direction then the roles of models are reversed. Since the i/o alphabets of remote tester and distributed tester differ due to synchronizing messages of distributed tester the behaviors are compared based on the i/o alphabet that is observable on SUT ports only. Second adjustment of models to be made for bisimulation analysis is the reduction of message propagation delays to uniform basis either on Δ or 2Δ in both models. Assume (due to closed world assumption used in MBT):

- the model for centralized testing is composed using parallel composition of SUT UTA model and remote tester UTA model: $M^{RT} = TA^{\mathtt{SUT}} \parallel TA^{r-TST}$

- The model M^{DT} for distributed testing consists of the parallel composition of local tester models and SUT model: $M^{DT} = TA^{\mathtt{SUT}} \parallel_{\parallel_i} TA_i^{d-TST}$ i = [1, n], n - number of ports locations.
- to unify the timed words $TW(M^{RT})$ and $TW(M^{DT})$ the equal communication delay between SUT and Tester is assumed.
- The SUT models in centralized and distributed tester cases are assumed to be equivalent and if they include control flow non-determinism then corresponding non-deterministic edges in both SUT models are mutually synchronized to avoid diverging reactions by SUT models.

Definition 3.1. (correctness of tester distribution): The mapping $M^{RT} \xrightarrow{Algorithm} M^{DT}$ is correct if TA^{r-TST} and $\parallel_{\parallel_i} TA_i^{d-TST}$ are observationally bisimilar, i.e. if TA^{r-TST} and $\parallel_{\parallel_i} TA_i^{d-TST}$ are respectively generating and accepting automata on common i/o alphabet $\Sigma^i \cup \Sigma^o$ then all timed words $TW(TA^{r-TST})$ are recognizable by $\parallel_{\parallel_i} TA_i^{d-TST}$ and all timed words $TW(\parallel_{\parallel_i} TA_i^{d-TST})$ are recognizable by TA^{r-TST}. Here, alphabet $\Sigma^i \cup \Sigma^o$ includes i/o symbols used at SUT-TESTER interfaces of M^{remote} and $M^{distrib}$.

Correctness verification of the distribution mapping:

Step 1: (Constructing generating-accepting automata synchronous composition):

- label each output action of TA^{r-TST} with output symbol a! and its co-action in $\parallel_{\parallel_i} TA_i^{d-TST}$ with input symbol a?;
- define parallel composition $TA^{r-TST} \parallel_{\parallel_i} TA_i^{d-TST}$ with synchronous i/o actions.

Step 2: (Bisimilarity proof by model checking): TA^{r-TST} and $\parallel_{\parallel_i} TA_i^{d-TST}$ are observationally bisimilar if following holds: $M^{RT} \models \mathtt{not\ deadlock} \wedge M^{DT} \models \mathtt{not\ deadlock} \Rightarrow TA^{r-TST} \parallel_{\parallel_j} TA_j^{d-TST} \models \mathtt{not\ deadlock}$ j = [1, n], n - number of local testers, i.e. the composition of bisimilar testers must be non-blocking if the testers composed with SUT model separately are non-blocking.

Solving Controllabilty and Observability Problems: The distributed test controllability problem arises when tester is not able to control the test execution over SUT, i.e. to emit test stimuli duly. Observability issues emerge when a local tester which sends act_{i+1} does not know whether act_i has been received by SUT via other location port. In centralized test architecture, tester can generate inputs only consecutively waiting each time for SUT output before sending next input for the result and continues then with the next set of inputs and outputs until the test scenario has been finished.

As will be shown in Lemma 3.1, Algorithm 1 generates the distributed testers that provides same test coverage as centralized remote tester.

Lemma 3.1. *(Equivalence of observable test traces)*
Let $TTraces(T^R)|_P$ and $TTraces(T^D)|_P$ denote timed traces mapped on ports in P of centralized remote tester T^R and the distributed tester T^D generated by Algorithm 1 *respectively. Then distributed tester T^D has same controllable test coverage as T^R, i.e. $TTraces(T^R)|_P = TTraces(T^D)|_P$.*

Note. Here we characterize the coverage in terms of test traces (mapped onto observable test ports) that are controllable by test inputs.

Proof. (by induction on the length of the sequence of i/o actions in trace $\pi \in TTraces(T^R)$)

Let T_r^D and T_s^D be the freely chosen local tester models at ports r and s ($r \neq s$), respectively and performing `i/o` actions act_i and act_{i+1}. Then the communication scenario $\pi \in TTraces(T^R)$ can consist of following cases:
Base case:
If no action has occurred in π then no action has taken place also in any of local testers T_j^D traces (π_1,..., $\pi_{|P|}$) mapped on ports $p_j \in P$, $j = 1, |P|$ because the initial conditions in centralized remote tester model T^R and in local tester models T_j^D are the same.
Inductive step:
Assume the coverage of test sequence π and sequences in (π_1,..., $\pi_{|P|}$)has been same up to i-th step of π then there are following cases:

case 1 : i-th step in π has been $act_i^O?$, then there are two possibilities of mapping it to distributed tester traces:

case 1.1 : $act_i^O?$ occurred in port p_r that is local to tester T_r^D

case 1.2 : $act_i^O?$ occurred in port p_s that observable by tester T_r^D remotely via tester's coordination action.

In Case 1.1 the action $act_i^O?$ is local event to T_r^D and by `Algorithm 1` the model T_r^D copies the edge with label $act_{i+1}^I!$ in the trace directly from centralized remote tester model T^R. More specifically, in Case 1.1 `Algorithm 1` substitutes the edge of T^R with label $act_i^O?$ with a pair of consecutive edges connected via a committed location where the first edge has label $act_i^O?$ and the second edge is labeled with broadcast channel label $act_i^{O*}!$ to communicate the event $act_i^O?$ to other local testers.

In Case 1.2 the edge with label $act_i^O?$ of centralized remote tester model T^R is substituted in T_r^D by `Algorithm 1` with synchronization action label $act_i^{O*}?$ that has co-action $act_i^{O*}!$ in local tester T_s^D which is triggered instantaneously after $act_i^O?$ has occurred at port p_s (symmetric with Case 1.1).

case 2 : i-th step in π has been $act_i^I!$, Similarly to Case 1, in Case 2 there are two sub-cases Case 2.1 where $act_i^I!$ occurs in the local tester that takes $i+1$ step of the trace π and Case 2.2 where actions $act_i^I!$ $act_{i+1}^I!$ occur at different ports. The proof of these cases is analogous to Case 1.1 and Case 1.2 respectively.

□

Example 3.2. Consider a faulty `SUT` test sequence and correct specification test sequence from remote tester shown in Figure 13

$RTS_{\mathbf{SUT}}$ = { in[1]! . (out[1],out[3])? . in[2]! (out[1],out[2])? . in[3]! . (out[1],out[3])? . in[1]! . (out[2],out[3])? . in[3]! . (out[1],out[3])? }

$RTS_{\mathbf{Spec}}$ = { in[1]! . (out[1],out[3])? . in[2]! (out[1],out[2])? . in[3]! . (out[1],out[3])? . in[1]! . (out[1],out[3])? . in[3]! . (out[2],out[3])?}.

After feeding RTS_{Spec} to algorithm 3.4, we get the following distributed test cases:

$Tester_1$ = { in[1]! . out[1]? . out[3]? . A_{Syn_21}**[in_[2]]**? . out[1]? . out[2]? . A_{Syn_31}**[in_[3]]** ? . out[1]? . out[3]? . in[1]! . out[1]? . out[3]? . A_{Syn_31}**[in_[3]]** ? . out[2]? . out[3]?}

$Tester_2$ = { A_{Syn_12}**[in_[1]]** ? . out[1]? . out[3]? . in[2]! . out[1]? . out[2]? . A_{Syn_31}**[in_[3]]** ? . out[1]? . out[3]? . A_{Syn_12}**[in_[1]]** ? . out[1]? . out[3]? . A_{Syn_31}**[in_[3]]** ? . out[2]? . out[3]?}

$Tester_3$ = { A_{Syn_13}**[in_[1]]** ? . out[1]? . out[3]? . A_{Syn_21}**[in_[2]]** ? . out[1]? . out[2]? . in[3]! . out[1]? . out[3]? . A_{Syn_12}**[in_[1]]** ? . out[1]? . out[3]? . in[3]! . out[2]? . out[3]?}

Each tester is waiting for coordination message A_{Syn_rs}**[in_[i]]** ? from other tester before applying any input to SUT, since we broadcast the coordination message (duplicates) to other local testers to synchronize the test runs at their local ports. This approach guarantees that each tester is aware of test execution status at other ports within Δ. Therefore, in $RTS_{\mathbf{SUT}}$ any controllability problems are avoided and detected if coordination message missing at Δ after expected occurrence of an i/o event at any other port. Thus, the observability issues can be detected locally at each port by executing local tester models in sync with other testers. Due to coordination messages the local testers can determine the input which is the cause of any output, therefore output faults are detected at locations where they are expected to occur. In the given faulty $RTS_{\mathbf{SUT}}$, the following part of test sequence has faults.

$RTS_{\mathbf{SUT}}$ = { in[1]! . (out[1],out[3])? . in[2]! (out[1],out[2])? . in[3]! . (out[1],out[3])? . $\overbrace{in[1]!.(\mathbf{out[2]}, out[3])?}^{\text{output fault 1}}$. $\overbrace{in[3]!.(\mathbf{out[1]}, out[3])?}^{\text{output fault 2}}$ }

Output fault 1 detection: As explained in `case 5`, the moment $Tester_1$ apply $in[1]!$, it synchronizes with other local testers by sending coordination messages and makes them aware that $in[1]!$ is sent to SUT and output may arrive. As the local testers are using replica of the same model, any wrong output receive will not conform to local tester model. Hence, output faults are detected. We proved that the synchronous messages added by `Algorithm 1` are sufficient to guarantee that distributed test architecture has the same coverage, control and observation power as centralized remote test architecture.

Algorithm analysis: We consider that the major drawback of this `Algorithm 1` is the extra communication overhead created by coordination messages among local testers. Each local tester representing the copy of original remote tester model is extended with auxiliary channels that are broadcasting the duplicates of locally observed events through channels between testers to other local testers to synchronize the test runs at their local ports. This results in extra communication overhead over the network.

3.4.2 Algorithm 2

We further optimize `Algorithm 1` in `Algorithm 2` by reducing the communication overhead, i.e. the amount of coordination broadcast messages by keeping only the messages that concern those local testers the progress of which depends directly on i/o actions of the others. The *Algorithm 2* implements the concept as following: each local tester needs to know only the occurrence of those i/o events at other ports that influence its behavior. Possible reactions of the local tester to these events are already specified in the initial remote tester model. The local testers project the global test sequence to that of observable on local ports of the SUT, and inter-location synchronization events are linked

to them. Due to multi-cast communication mode, it is possible to reduce the coordination overhead by sending synchronization messages to other testers selectively so that only the synchronization signal is sent to those testers which further action depends on it. Details of the algorithm are presented in Algorithm 2.

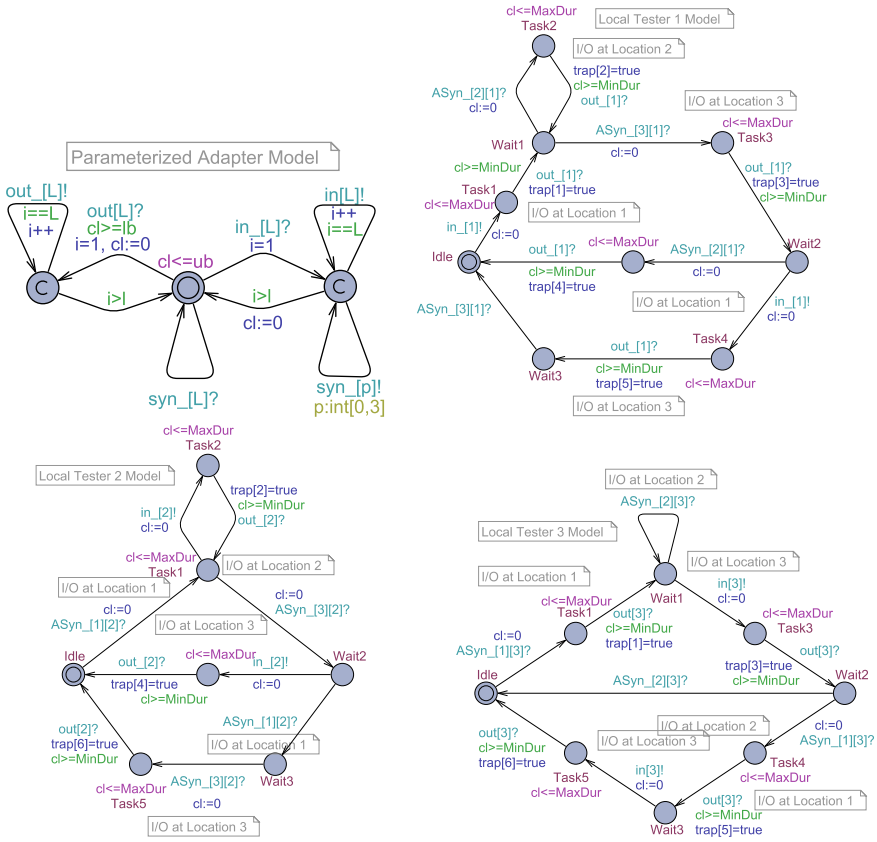


Figure 15 – Parameterized Adapter and Local Tester Models.

Algorithm Description: Line 4-11 adds an adapter automaton to each local tester instance. The purpose of adding an additional adapter instances to each local tester instance is that it synchronizes the local communication between SUT local ports and a local tester with other testers in different locations. Its model is derived from remote tester model by adding original channels of SUT and by renaming channels of local testers. For clarity, notations T_l and A_l represents local tester and local adapter respectively; T_o and A_o represents the tester and test synchronization adapter at other ports respectively.

Note: To avoid terminological confusion, we use in this section term "adapter" when referring to the test synchronization adapter not to the "test adapter". While test adapter interfaces test model and real SUT *the test synchronization adapter automaton serves only to synchronize between local tester models.*

The channel $\texttt{in}[l_n]$ denotes the input at location l_n, E represents Emission of `chan` and R represents Reception of `chan`. The $\texttt{in\_}[l_n]$, $\texttt{out\_}[l_n]$ are the channels between local synchronization adapter automaton adapter and local tester automaton. The $chan\ \texttt{in\_}[l_n]\texttt{?}$ represents the reception R of input i.e. $T_l \xrightarrow{\texttt{in\_}[\texttt{l}_\texttt{n}]\texttt{?}} A_l$ and $chan\ \texttt{out\_}[l_n]\texttt{!}$

Algorithm 2 Automated Construction of Adapter and Local Testers

input: M^{RT};
output: $||_n M^{\mathrm{DT}}\ n \in \mathbb{N}$;
//build a new model (adapter) by adding below channels
for all $l_n \in Loc(\mathrm{SUT})$ **do** ▷ $n \in \mathbb{N}$
    Add chan `in_[`l_n`]?` ▷ R: $T_l \rightarrow A_l$
    Add chan `in[`l_n`]!` ▷ E: $A_l \rightarrow \mathrm{SUT}$
    Add chan `out[`l_n`]?` ▷ R: $\mathrm{SUT} \rightarrow A_l$
    Add chan `out_[`l_n`]!` ▷ E: $A_l \rightarrow T_l$
    Add chan `sYn_[`l_n`][`l_p`]!` ▷ E $A_l \rightarrow A_o$, $p : int[1,N]$
    Add chan `sYn_[`l_p`][`l_n`]?` ▷ R $A_l \rightarrow A_o$
end for
//construction of local tester models
copy M^{RT} to M^{l_n} ▷ take clone at each location
for all $M^{l_n}, n \in \mathbb{N}$ **do**
    for all `chan[`l`]: in[`l_n`]/out[`l_n`]` pairs $\in M^{l_n}$ **do**
        Case 1:
        if edge.`in[`l_n`]` $\wedge$ edge.`out[`l_n`]` $\wedge\ l_n \in P_{l_n}$ **then**
            Rename `chan in[`l_n`]!, out[`l_n`]?` to `in_[`l_n`]!, out_[`l_n`]?`
            for all edge.`out[`l_i`]` $\in Loc(\mathrm{SUT}) \wedge i \neq n$ **do** ▷ *output for other ports*
                Remove `chan out[`l_i`]?`
            end for
        end if
        Case 2:
        if (edge.`in[`l_i`]` $\wedge\ l_i \notin P_{l_n}$) **then** ▷ $i \neq n$
            Replace `chan in[`l_i`]` to *chan* `sYn_[`l_p`][`l_n`]?`
            if (edge.`out[`l_i`]` $\wedge\ l_i \in P_{l_n}$) **then**
                Rename `chan out[`l_i`]?` to `out_[`l_i`]?` at $P_{l_i} \in$ `out[`l_i`]`
            end if
            if (edge.`out[`l_i`]` $\wedge\ l_i \notin P_{l_n}$) **then**
                Remove edge.`out[`l_i`]?` from P_{l_n}
            end if
        end if
        pass control: to $M^{l_i} \in$ `in[`l_i`]!`
        repeat Case 1.
    end for
end for

represents the emission E of output i.e. $A_l \xrightarrow{\texttt{out\_[ln]!}} T_l$. Similarly, the channels in[l_n]? out[l_n]! are the channels between SUT and synchronization adapter. In order to coordinate with testers at other ports through adapter, chan sYn_[l_n][l_p]! is added to the adapter model in Line 9-10, local adapter sends sYn_[l_n]! to each adapter and synchronize with its co-action at other local adapter sYn_[l_p][l_n]?.
Now, the construction of local testers, for each port locations l_n, we take clone of centralized remote tester model M^{RT} to be transformed into a location specific local tester instance M^{l_n} (Line 13). The loop in Line 15 says for each clone testers model M^{l_n}, we go through all the edges i/o pair. For clarity, we divided the distribution into two cases, in Line 17, *Case 1* says if the edge has a synchronizing channel i.e in[l_n]/out[l_n] and the channel belongs to same port location $l_n \in P_{l_n}$ then we *Rename* the *chan* in[l_n]!, out[l_n]? to in_[l_n]!, out_[l_n]? as shown in Figure 15 and *Remove* the output channels out[l_i]? generated in response to in[l_n]! where i $\notin$ n. Basically, idea is to minimize the automata M^{l_n} by removing all synchronizing channels that do not belong to local actions at this location.

Case 2 In line 24, if input chan edge.in[l_i]! where i $\neq$ n, does not belong to P_{l_n}, *Replace* chan edge.in[l_i]! with sYn_[l_p][l_n]?. This helps to minimize the unnecessary channels from local tester which do not belong to particular port. Also, adding channel sYn_[l_p][l_n]? avoids controllability and observability issues. For instance, this case can have two major subcategories: (i) edge.in[l_i]! where input does not belong to P_{l_n} but it produce output edge.out[l_m]! where m = n, belongs to P_{l_n}. In this case, we rename the channel out[l_m]! to out_[l_n]?. (ii) edge.in[l_i]! where input does not belong to P_{l_n} but it produces output edge.out[l_m]! where m $\neq$ n, does not belongs to P_{l_n}. We remove the channel from automaton.

Finally, repeat the similar steps for other local testers. Figure 15 represents the generated local testers with corresponding parameterized synchronization adapter model where parameter L denotes the geographical location.

By applying Algorithm 2 it is possible to reduce the coordination overhead by sending synchronization messages to other testers selectively so that only the synchronization signal is sent to those testers which further action depends on it. It presumes preliminary static analysis of the model to extract the dependencies between actions that may follow each other in the timed traces. Since standard algorithms for dependency analysis [67] can be applied for this task we refer its detailed discussion in the Thesis.

Theorem 3.1 *(Distributability of centralized remote tests)* Let T^R and T^D denote Uppaal timed automata models of a centralized remote tester and distributed tester derived from T^R (by using distribution algorithms), respectively and let $TTraces(.)|p$ denotes the projection of $TTraces(.)$ onto test ports P, then $TTraces(TT^R)|p = TTraces(T^D)|p$ implies T^R tioco T^D and T^D tioco T^R.

Proof. By Lemma 3.1 we conclude that Algorithm 1 and Algorithm 2 assure the equivalence of timed traces $TTraces(T^R)|p = TTraces(T^D)|p$. By Lemma 3.1 and definition of tioco (Impl **tioco** Spec $\equiv \forall \rho \in$ TTraces(Spec) : Out(Impl After ρ) $\subseteq$ Out(Spec After ρ)) we can conclude that $TTraces(T^R)|p = TTraces(T^D)|p$ implies T^R tioco T^D . Similarly by symmetry of equivalence $TTraces(T^R)|p = TTraces(T^D)|p$ the conformance T^D tioco T^R follows. □

3.5 Δ - Delay Controllability

While `Lemma 3.1` and `Theorem 3.1` stated the equivalence of observable traces and conformance relation between remote centralized tester and distributed tester derived from it by `Algorithm 1` and `Algorithm 2`, the proofs were based on the assumption of neglecting real message propagation delay Δ. The delta controllability correctness is expressed by the following `Proposition 3.1`.

Proposition 3.1. *A model-based remote tester that satisfies 2Δ delta test controllability requirement can be transformed to an observationally equivalent set of distributed and internally coordinating local testers which satisfies (one) Δ controllability requirement, provided the coordination latency does not violate the model original timing constraints.*

From the hypothesis and definitions explained in Section 2.2.1, if the remote tester wants the SUT to receive input at global time it must satisfy the timing constraints:

$$\overbrace{g(v) \wedge Inv_{l'}(v')}^{\text{clock constraints at tester}} + \overbrace{\mathtt{d}_\Delta}^{\text{propagation latency}} \leq \mathtt{W}^{max}_{SUT} \tag{1}$$

And to receive outputs from SUT, tester must satisfy:

$$\overbrace{\mathtt{C}_{SUT}}^{\text{computation time of SUT}} + \overbrace{\mathtt{d}_\Delta}^{\text{propagation latency}} \leq \mathtt{W}^{max}_{st} \tag{2}$$

By Definition 2.1, the clock constraint $g \in \mathtt{G}(\mathbb{T})$ is satisfied by the clock valuation $v \in \mathbb{R}^{\mathbb{T}}_{\geq 0}$ and invariant of target location $Inv_{l'}$ is satisfied by v' where the clock valuation v' is obtained by applying clock update $u \in \mathtt{U}(\mathbb{T})$ on v. From equation 1, we conclude that if the remote tester wants the SUT to receive input by some deadline it must send the necessary inputs $\mathtt{P}_\Delta$ earlier (because of latency) so that SUT receives it within the expected timing window. Similarly, by inequation (2) SUT sends output and remote tester receives it at time $(\mathtt{C}_{SUT} + \mathtt{P}_\Delta)$ which must be not greater than maximum waiting time of tester $\mathtt{W}^{max}_{st}$.

The communication pattern modification due to distributed test architecture eliminates the message propagation time between the local tester and SUT. We assume that, communication delay between a local tester adapter and the SUT port is negligible and one Δ communication delay among local testers which is independent of SUT. Using *hypothesis* 1 and 2, if the local tester wants the SUT receives input timely it must satisfy the following timing constraints:

$$\overbrace{g(v) \wedge Inv_{l'}(v')}^{\text{clock constraints at tester}} \leq \mathtt{W}^{max}_{SUT} \tag{3}$$

From the assumptions defined in Section 3.2, each local tester react to two kinds of actions: coordination message from other local testers and output action from the SUT. We assume that these two actions must be received by each tester in less than the maximum waiting time of tester. This is essential and adequate (amount of time testers has to wait for reception) before concluding any verdict whether SUT conforms to $\mathtt{S}pec$, and it is true only if $\mathtt{C}_{SUT}$ and propagation time $\mathtt{P}_t$ are within certain time bounds. Therefore,

each local tester must satisfy:

$$\overbrace{C_{SUT}}^{condition1^*} + \overbrace{\lceil P_\Delta \cdot \Sigma_{i=1}^{n}(M_{Syn}) \rceil}^{condition2^*} \leq W_{tt}^{max} \tag{4}$$

$Condition1^* : T_E - T_R \approx negligible$

$Condition2.1^* : act_q \in Output_i \;\&\; P_{tt}^{max} \leq ClockTick(T_{ub})$

$Condition2.2^* : act_q \in Input_i \;\&\; P_{tt}^{max} \leq ClockTick(T_{ub})$

Since the local testers are attached directly to the ports of SUT and the message propa-

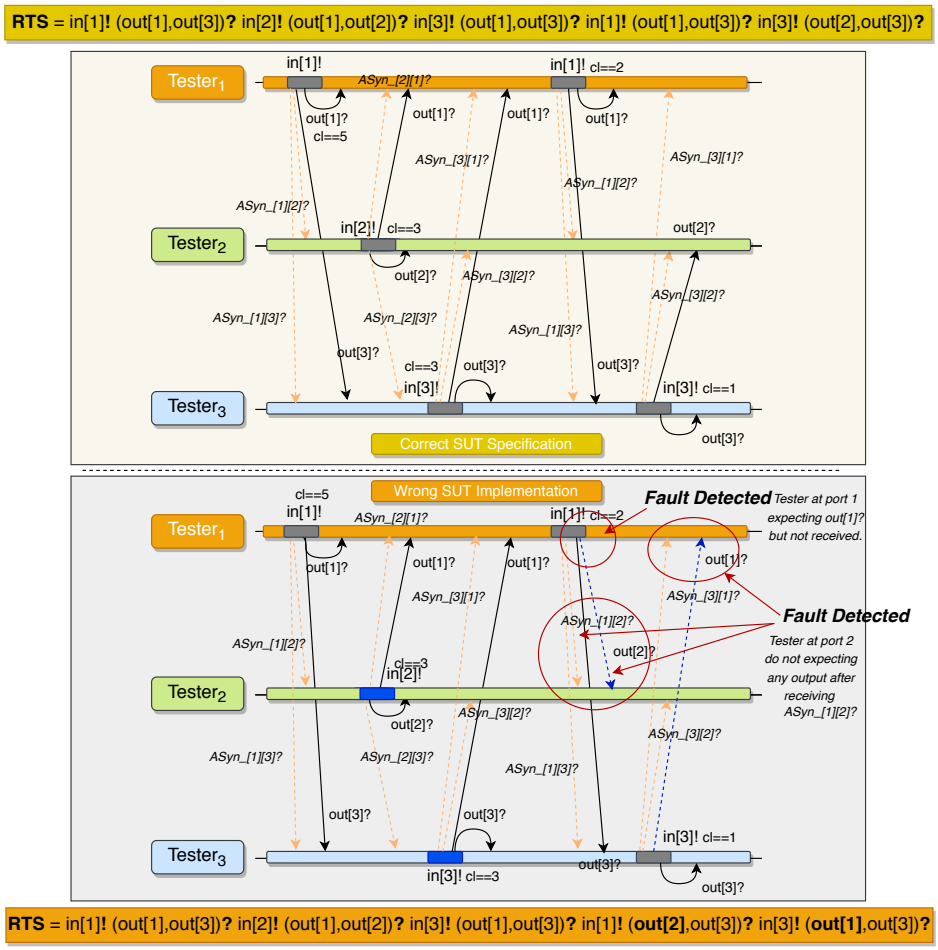


Figure 16 – Algorithm Performance

gation time and computation time are assumed to be negligible. Hence communication delay at SUT output is also considered negligible as expressed in *Condition 1.* Each local tester may block its test execution at step which depends on the occurrence of next

action on its port. We divided *Condition 2* further into important two cases. First, we assume that there is no restriction on the order of receiving simultaneous outputs from SUT and coordination message from other local testers but the coordination messages about the arrival of their locally observable SUT outputs must be received by other local testers within maximum waiting time of each local tester. In our case maximum waiting time of each local tester is assumed to be one clock tick as shown in clock tick gen model.

From inequations (1) and (2), since there is latency between tester and the SUT, tester should not wait to receive outputs before sending the input to SUT. Hence 2Δ-condition is sufficient for satisfying timing correctness of the test. From conditions (3) and (4), we conclude that, in order to guarantee one Δ controllability of distributed local testers, they have to respect timing constraints given in inequation (4). Since the local testers are attached to the test port directly the communication delay between a local tester adapter and the SUTport is negligible (one P_Δ eliminated) and only one P_Δ (which should be less than W_s^{max}) is required to coordinate the test activity.

Solving Controllabilty and Observability Problems: By considering all inequations discussed above, we demonstrate Algorithm 2 performance where local testers communication helps to overcome distributing testing issues as shown in Figure 16.

3.6 Chapter Summary

In this chapter, we have presented the main results of thesis. We presented the test architecture for testing distribute system and discussed its components such as adapter, coordination between adapter and local testers. We have presented several hypothesis and definitions required to present our approach. We have also discussed in detail the issues in distributed testing and illustrated them with examples. We introduced centralized tester distribution algorithms and discussed their improvements in terms of performance. We presented test scenario where a centralized testing cannot be applied. The experiments show that the distributed test architecture is more scalable and efficient in terms of test reaction time (assuring Δ controllability) than centralized remote test architecture. We also demonstrated how synchronizing tester models is capable of detecting controllability and observability issues in faulty SUT. The major drawback of Algorithm 1 is the extra communication overhead created by coordination messages exchanged among local testers. Each local tester model copying the local activities of the original remote tester model is extended with auxiliary channels that are broadcasting the duplicates of locally observed events through inter-tester channels to other local testers to synchronize the test runs at their local ports. This results in extra communication overhead over the network, that we further optimize in Algorithm 2 by reducing coordination broadcast messages with targeted messages that concern only those local testers the progress of which depends on remotely observed i/o actions. The *Algorithm 2* implements the concept where each local tester needs to know only the occurrence of those i/o events at other ports that influence its behavior. The usage of tester distribution algorithms is demonstrated further in Chapter 4 where they are discussed in the context of an industry scale case study.

4 Case study: Testing Flexibility Contracts for Ancillary Services in Energy Grids

4.1 Chapter Overview

In this chapter, we demonstrate the elaborated distributed testing approach on industrial case study. We present the short description of main concepts such as energy flexibility, flexibility load and contracts, buildings energy management systems (BEMS), automatic gain controllers (AGC), rate of frequency change (ROCOF) etc. and their roles in SUT. We demonstrate how industrial requirements can be transform into model (SUT and Remote Tester) and used to generate local testers. We apply the proposed test architecture using MBT platform DTRON for facilitating distributed testers deployment and execution.

4.2 System Description

Electric grids are prototypical examples of CPS having interdependent physical and cyber components. Current practice to operate energy grids reliably requires matching generation with demand at all times. In effect, this reduces the energy efficiency of the generators, diminishes the benefits of the renewable sources, and increases energy cost. An alternative is to leverage the presence of flexible loads, i.e. components whose consumption or its pattern can be changed. The Heating, Ventilation and Air-Conditioning (HVAC) systems by arbitraging around their comfort bands can provide flexibility to the energy grids. The use of HVAC systems for providing ancillary services (frequency regulation) to energy grids is a recent research topic. Much of the existing results is mainly focussed on computing the flexibility by solving a large optimization problem, whose scalability is restricted by the size of the building.

To guarantee reliability such flexibility computation algorithms should be verified for timing performance. However, the distributed nature of the flexible loads and fast time frames emerge as a key challenge in verifying their timing performance. Centralized approaches for testing such systems often do not meet timing constraints and therefore, methods wherein the functionality of the centralized remote tester should be distributed at the local nodes providing flexibility as required. Our motivation here is to validate and demonstrate the usability of the distributed testing method by applying it on SUT where flexibility contracts in energy grids are providing ancillary services.

Flexible Load: The flexible load considered in this investigation is the HVAC system in the commercial buildings. It has a centralized air-handling unit (AHU) that supplies air to individual zones and there is a chiller that supplies cooled water to the plant that passes through a heat exchanger to absorb the latent heat to provide the cooling. The HVAC consumption is mainly due to the fan and chiller coil. The fan consumption can be changed by varying the speed using a speed drive.

The Baseline Contract: This investigation considers the commercial building that has a HVAC system supplying air to individual zones. These zones are controlled by a model predictive controller that aims to reduce the energy cost without violating the thermal dynamics of the building, comfort margins provided by the user and temporal constraints that model the user preferences. The energy cost of the zone is composed of two parts: fixed and variable time-of-use charges that vary depending on the time of the day. The costs are generally published 24 hours in advance for a particular day based on market clearing prices in the day-ahead markets.

Flexibility Contracts The method of contracting flexibility in our investigation is based on the market-based control, a distributed control strategy wherein a virtual market creates competition among agents for "commodities", such as electricity power or cooling/ heating energy. Each of the entities in the transaction are modelled as a self-interested agent trying to maximize its benefit. In our case, there are four agents: the utility, buildings, market and building energy management system as shown in Figure 17.

The *market agent* is a virtual one that is created to provide competition between agents for commodities i.e., cooling energy and electric power in our case. The *market agent* is designed to have a sort of double-blind auction, in which each agent bids for a flexibility curve and not for a single price-quantity pair. The zone agent models the variable-air-volume (VAV) and they bid demand curve defined by three points: minimum desired cooling rate (maximum desired set-point), maximum desired cooling rate (minimum desired set-point) and baseline (optimal computed for a given temperature band). To avoid discomfort to the user the minimum and maximum deviations from the set-points are bounded. In addition, cooling supplied to the zone is also bounded by the flow capacity of the VAV. Therefore, the minimum and maximum flow-rate of VAV terminal box can be used to calculate the absolute boundaries for the cooling demand curve submitted by each zone agent. To compute the cooling energy required, the zone agent uses the predictions on cooling energy.

The *building energy management system* (BEMS) agent provides the cooling power required for solving an optimization problem with the baseline contract as the energy requirement. In addition, it provides the upper and lower bound on the total energy consumption using the demand requests placed by the zones. To compute the optimal cooling energy distribution, it considers the air-handling unit constraints and uses the optimization model to compute the flexibility provided by each zone for a given time-interval τ_k. This is transmitted back to the zone controller as accepted nominal, up-and-down flexibility allotment for each zone. This aggregate of the base-line, maximum and minimum flexibility is used to define the market clearing prices for the flexibility curve. These agents deployed for performing market-based control work in two-modes: 1. off-line mode and 2. On-line mode.

In the off-line mode, the zones compute their flexibility and time-slots in which the flexibility is available by solving an optimization problem. Once the flexibility is computed, the base-line contracts which involve the optimal consumption details of the zones are informed to the BEMS. The external inputs required during off-line mode are sent by the BEMS for the duration it senses the demand to be high. The triggers sent during off-line mode are given by pairs (5)

$$\mathscr{T} = \{\tau_k, \gamma_k\} \tag{5}$$

where τ_i and γ_i indicate the time-slot and trigger condition for a particular time-slot of an hour, respectively. Based on the trigger $\mathscr{T}$, off-line the baseline contracts are published which minimizes the cost of the cooling requests. In addition, the zones also publish the flexibility available during each time-slot to the central controller by solving the optimization model with the temperature bounds that they can arbitrage (up/down) for providing flexibility.

4.3 Online Operation Scenario

In the on-line mode, the *utility agent* that models the automatic gain controller (AGC) sends triggers when it senses that there is a need for additional reserves due to rate-of-change of frequency (ROCOF) being critical as shown in Figure 17. This is an external trigger

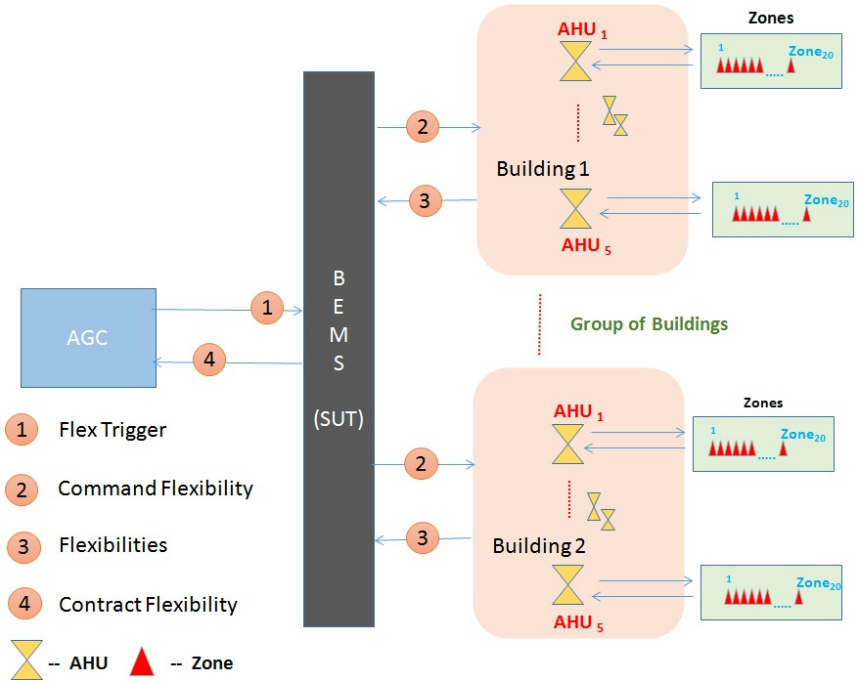


Figure 17 – Distributed flow of generating flexibility contracts in online mode

to the SUT. On receiving the triggers, the BEMS agent requests for bids from zone agents. The zones on receiving the requests forward their flexibility bids to the BEMS by solving The aggregated flexibility bids from the zones are published in the market by the BEMS agent. The AGC then accepts the bid based on the cost (lowest bid is accepted first) from different zones. The sequence of operations is explained in Figure 17 by sequencing the interactions between agents.
An important criteria for harnessing flexibility for providing ancillary services to the grid is that the computations should be completed within a pre-specified time bound and the VAV should be commanded to provide the flexibility. Verifying the timing performance with a centralized tester for multiple buildings may not be feasible due to unability to apply scenarious where multiple interactions with SUT occur simultaneously. A better alternative in this case is to distribute the testing to remote testers, i.e. zone agents. The numerical values of the computation time shown in Figure 18 are used for equipping the test model with timing information. The sequence of operations is explained step by step as follows:

- *Step 1*: When there is a ROCOF (rate of change of Frequency) an external input occurs. The Automated Generation Control (AGC) sends triggers to the BEMS for getting flexibility.

- *Step 2*: The BEMS commands the flexibility in the online mode by changing the power supplied to the fan or changing the Variable Air Volume (VAV) .

- *Step 3*: The BEMS agent requests for bids from zone agents. The zones on receiving

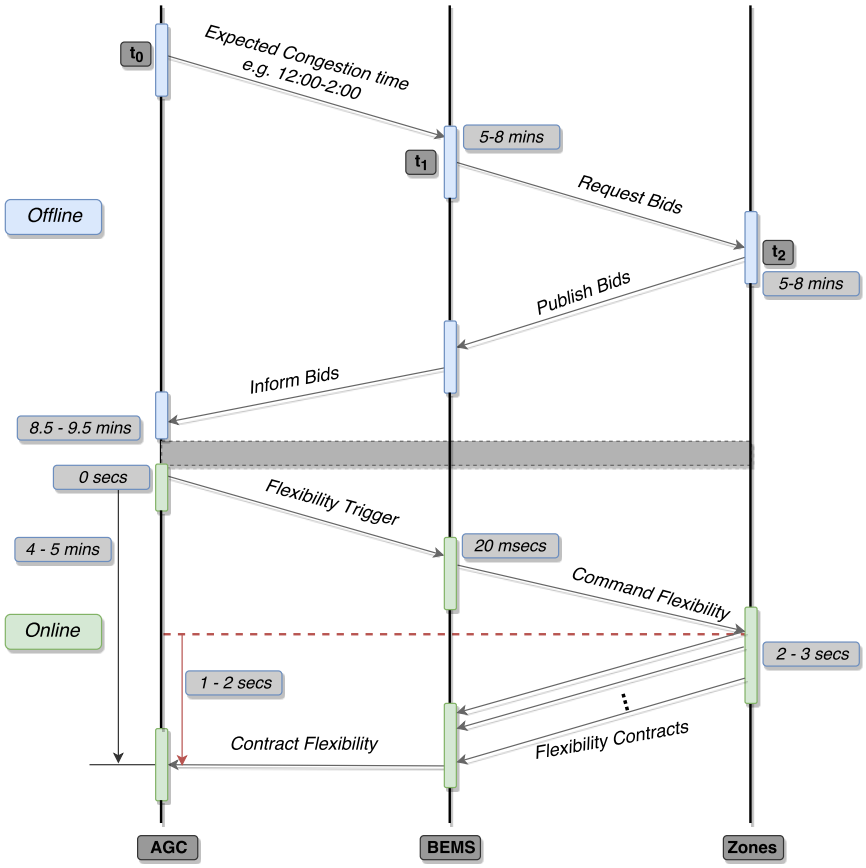


Figure 18 – Sequence of Operation

the requests forward their flexibility bids to the BEMS by solving the zone optimization constraints.

- *Step 4*: The contracted flexibility is informed to the Utility (AGC), i.e., contracts are published.The contracts are accepted until the ROCOF is done away with based on the merit (low cost flexibility is accepted first)

The computation times of described steps shown in Figure 18 are for 100 zones. For more than 50 zones, the current algorithms practically do not work due to computation complexity reasons. We need to make use of the token algorithm in which only the offline computation phase may differ, but the online computation should remain the same.

4.4 Formalization of Distributed SUT Observable Behavior and Centralized Remote Tester

We start modelling the test scenario, at first, with specifying the behavior at i/o ports of SUT model shown in Figure 19. In distributed testing architecture all distributed location

i/o ports of AGC, BEMS and Zones all together form the SUT. Each building may typically contain 5 AHU and each AHU has 20 zones, which means each building has 100 zones of medium scale.

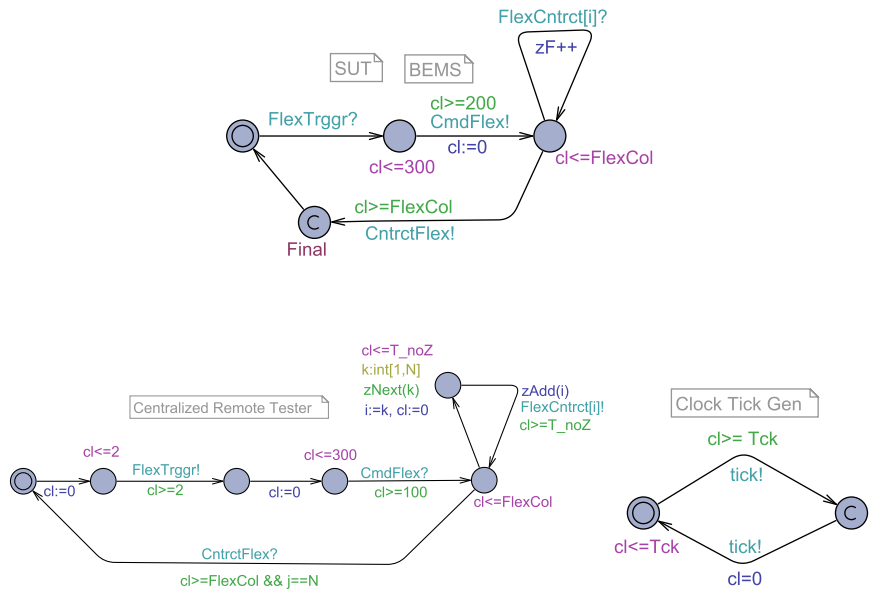


Figure 19 – SUT and Centralized Remote Tester Model.

4.4.1 Specification of Test Purpose

We assume in current test case that the interaction between BEMS and multiple zones is based on the protocol according to which all zones are sending their flexibility contracts depending on the number of zones in AHU either:

- simultaneously;
- at random time instances;
- deterministically using some token passing protocol (number of zones is > 50);

Table 3 – The Flexibilities Provided By Each Zones

Case 1:	int zHold_Flexs[N+1]={0,5,5,5,5,5,5,5,5,5,5};
Case 2:	int zHold_Flexs[N+1]= {0,5,2,5,3,5,5,5,7,5,10};
Case 3:	int zHold_Flexs[id_zB] = {2,5,5,5,2,5,1,5,1,5};

Three test scenarios are studied:

- *Scenario 1*- All the zones which offer flexibility in a given time interval, use the choice by merit (the zone with lowest bid is engaged first). As shown in Table 3 and in Table 4, Cases 1 and 2 are used to show that their sum of flexibilites by different zones satisfies the ROCOF. Note, numbers presented in the tables are encoded in UTA model arrays of type integer as arrays of flexibilites and bids.

Table 4 – The Flexibility Bid Provided By Each Zones

Case 1:	int zBids[N+1] = {0,1,1,1,1,1,2,1,1,2,1};
Case 2:	int zBids[N+1] = {0,1,1,2,1,3,1,2,1,3,1};
Case 3:	int zBids[N+1] = {0,0,2,1,1,1,3,0,1,0,1};

- *Scenario 2*- Not enough flexibility bids, increases the incentives (motivators) to obtain the flexibility. As shown in Table 3, 4, Case 3 used to show sum of flexibilites by different zones less than ROCOF

- *Scenario 3*- Zero bids, then the system command the generator or secondary reserves (Solar panels, storage etc.)

4.4.2 Test Case for Remote Tester

The goal of the test case is to sample the above scenarios, we specify the test purpose as:

$$\mathtt{E} <> \mathtt{SUT.send\_Cntrct\_Flex} \;\&\&\; \mathtt{gCl} <= \mathtt{Stopwatch.Passtext}$$

The test sequences and simulation of the SUT and $\mathtt{M}^{\mathtt{RT}}$ are shown in Figure 20, 21 respectively. The traces are generated with Uppaal model checker option *fastest*.

4.5 Distributed Local Testers and Test Purpose

To demonstrate the usability of proposed testing approach we apply *Algorithm 2* for generating the distributed tester. After applying the *Algorithm 2*, the centralized remote testing architecture depicted in Figure 19 is transformed into a set of distributed testers and extended with their coordination adapters. The corresponding local testers composed with their adapter model are depicted in Figure 22. As discussed in `Chapter 3`, this results in reducing the tester reaction time and enables testing a real-time distributed system under the timing constraints close to the message propagation time range.

4.5.1 Test Case 1 for Distributed Tester

The goal of the test case is sample the Scenario1. We specify the test purpose as TCTL query:

$$\mathtt{E} <> \mathtt{SUT.send\_Cntrct\_Flex} \;\&\&\; \mathtt{gCl} <= \mathtt{Stopwatch.Passtext}$$

The test sequences of the SUT and $\mathtt{M}^{\|\mathtt{DT}}$ are shown in Figure 23. The traces are generated with Uppaal model checker option *fastest*.

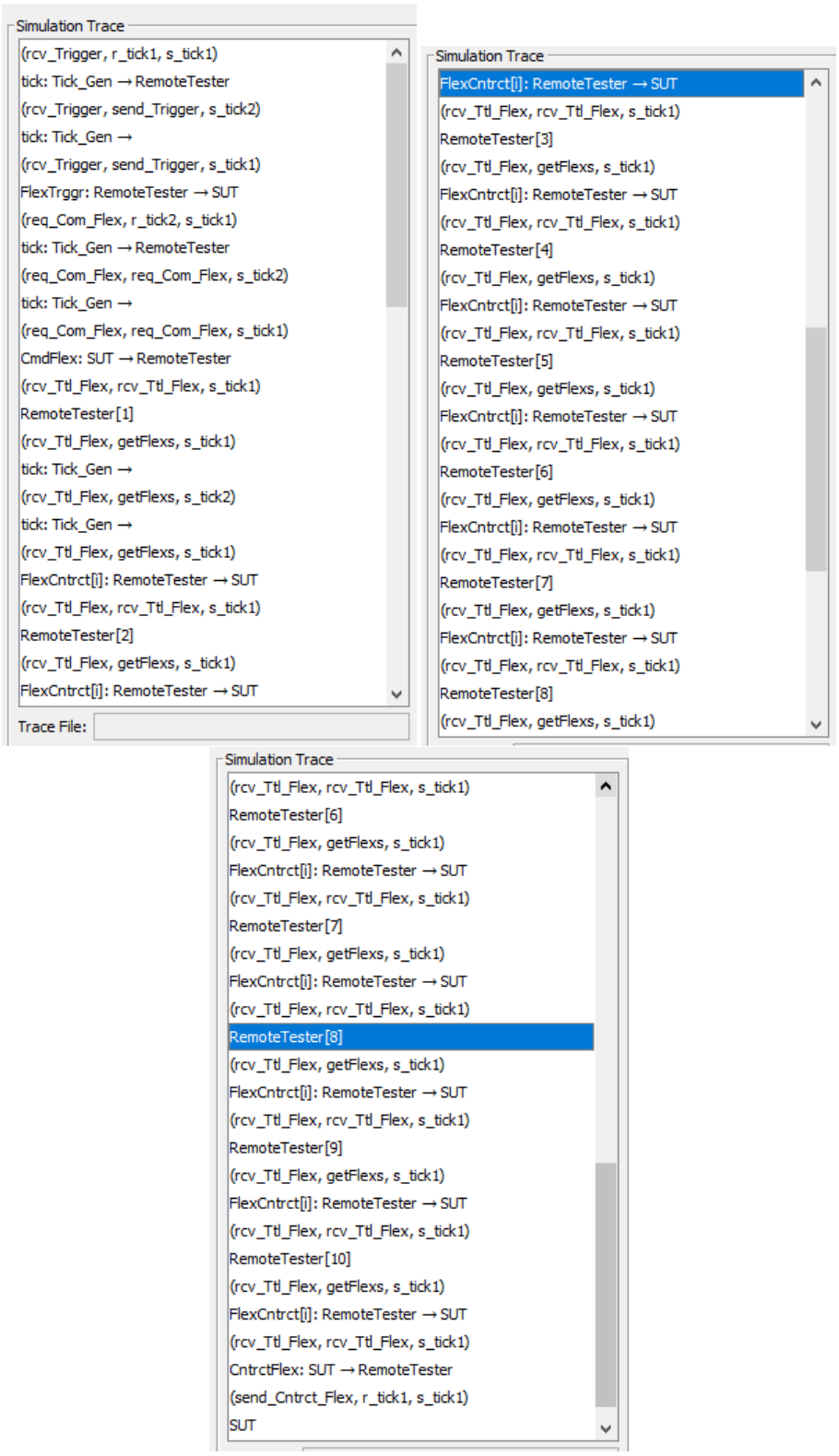


Figure 20 – Fastest Trace that satisfy the test purpose

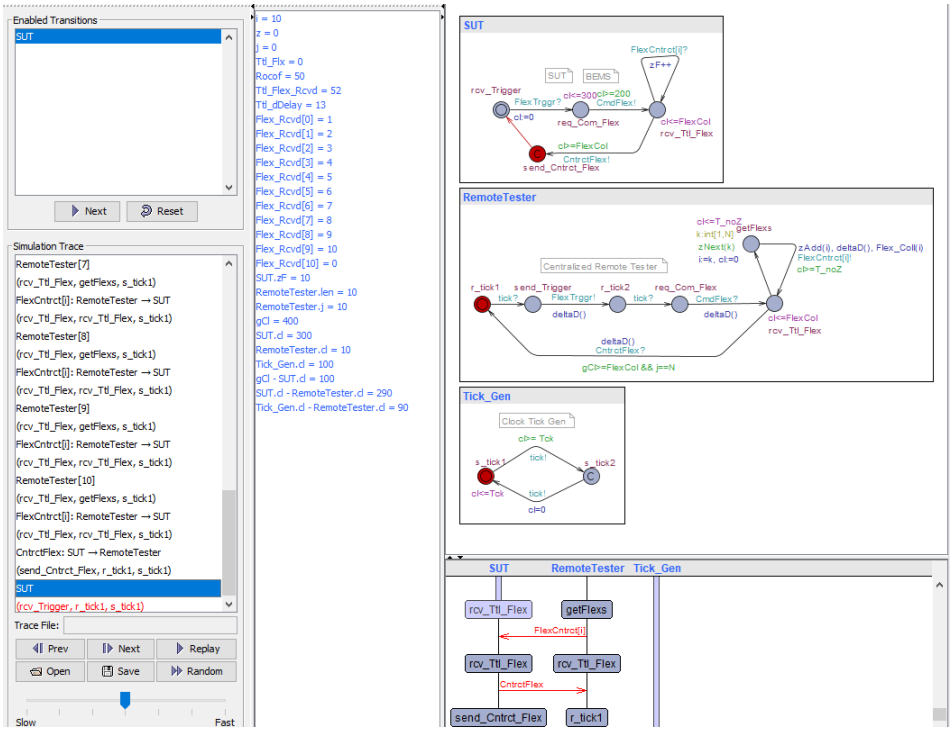


Figure 21 – Full Simulation View of SUT and M^{RT}

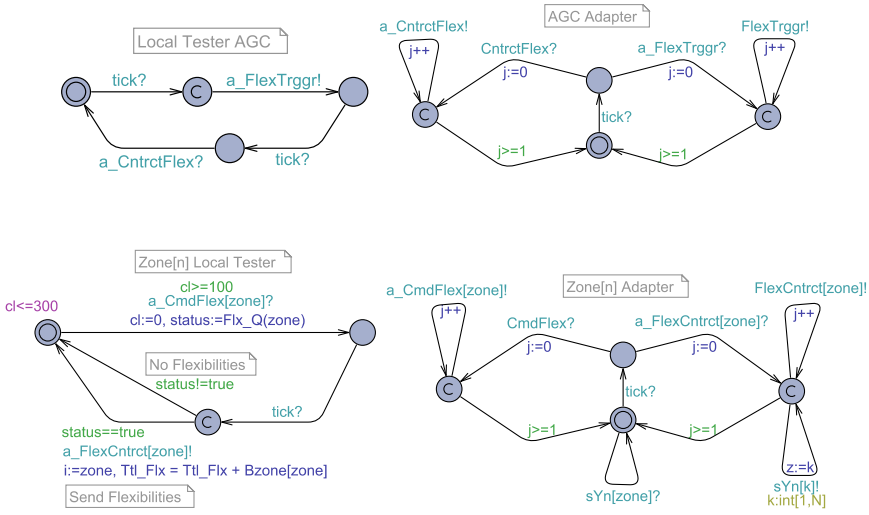


Figure 22 – Parameterized Distributed Local testers and Adapter Models.

Figure 23 – Full Simulation View of SUT *and* $\mathtt{M}^{\mathtt{DT}}$

4.6 Experiment Setup

4.6.1 Distributed Test Execution with DTRON

For testing the distributed ancillary services in energy grid a distributed testing tool DTRON [8] is used. A DTRON instance running at one port serves as a local tester execution engine and the publish–subscribe messaging allows the observation of a global trace. Figure 24 shows a conceptual view of the distributed runtime deployment configuration of DTRON. From bottom up, there is a SUT or a set of distributed SUT components that have test ports. Each port used in the test is directly connected to a DTRON instance running against it. This instance can be an Adapter, a Model execution engine, or a combination of both. DTRON instances communicate over Spread [9], which can be clustered. The local tester

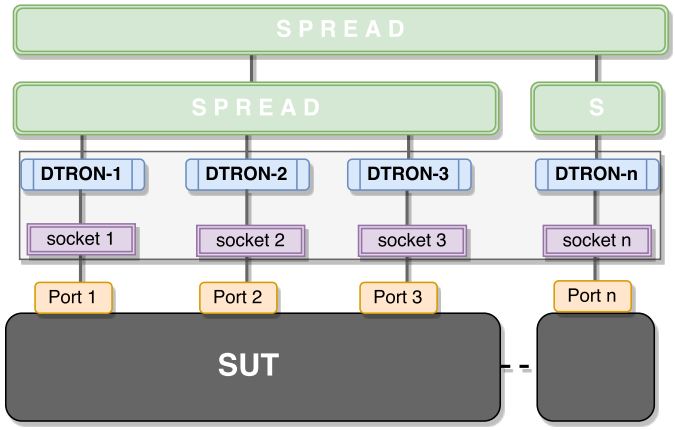


Figure 24 – Distributed testing data-flow in DTRON.

models embedded into DTRON instances are interfaced to SUT ports via test adapters and subscribed to their corresponding Spread broker. There can be many brokers while preserving the correct message serialization over all brokers. DTRON binds its communication socket to a specific broker to publish and subscribe for messages. Spread takes care of the network route discovery and planning. So, a message published to one broker can be received by a subscriber to another broker in another network segment. Uppaal TRON uses the socket based interface for API integration since it provides support for Java integration and for virtual clocks. These are the mechanisms to agree on how model time passes and allow for delta-testability. The usage of spread addresses also the problem of non-uniform message transmission delays between distributed DTRON testers. Consider the Figure 24, we would normally expect that if DTRON-1 publishes a message at time point t_1 and DTRON-2 after that at t_2, then $t_1 < t_2$. However, if DTRON-1 exhibits an internal delay longer than DTRON-2, it could happen that t_2 is actually published before t_1 and therefore $t_2 < t_1$. This may lead to a conformance violation with the model. But with DTRON we can measure the delays at the adapter level and use virtual time to agree that $t_1 < t_2$ even if by receiving side observations it was $t_2 < t_1$ instead. We refer to Δ as the time interval during which we allow events to be swapped in this manner.

Experiments and DTRON working are shown here[2]

[2]https://www.youtube.com/watch?v=4_JLrSojUjE

4.7 Chapter Summary

In this chapter, we demonstrated the feasibility of approach on an industrial case study. We presented the short description of main concepts such as energy flexibility, flexibility load and contracts, buildings energy management systems (BEMS), automatic gain controllers (AGC), rate of frequency change (ROCOF) etc. and their roles in SUT. We demonstrated how industrial requirements can be transform into model (SUT and Remote Tester) and used to generated local testers. We demonstrate the proposed test architecture using MBT platform DTRON as a test execution platform used for facilitating distributed testers deployment and management.

5 Conclusions

In this thesis we have presented a model-based distributed testing methodology, architecture and provably correct test development for time critical distributed systems. The presented approach aims at online model-based testing of real-time distributed systems possibly with non-deterministic behavior.

It has been shown that proposed approach satisfies better the timing constraints faced in solving controllability and observability issues which are often limit the use of the centralized testing architecture. The thesis have sought for solutions also to the issues such as *timeliness*, *latency*, *observability*, *controllability*, *reproducibility* and *non-determinism* "in the context of testing real-time distributed systems.

As for broader context of distributed testing, the early works on MBT have been reviewed and it is concluded that they focused mostly on testing distributed non real-time systems. However, very few of them adapted model-based testing as standard testing techniques. Most model-based testing techniques lack empirical evaluation for industrial environment and provide only partial tool support.

To address these issues and meet the demand from academic and industrial perspective, we proposed a complete pipeline starting from modelling *n-ports* distributed systems specification, development of distributed tester (using novel algorithms), deploying synthesized testers to local ports and executing tests. The proposed approach is outlining following novel results:

- *Test Model Development*: We introduced a semantic foundation for modelling *n-ports* distributed systems with Uppaal timed automata as a relevant formalism to represent time-critical behavior. We show how the SUT and remote tester models are built in a step-wise manner. We showed how timing requirements and non-deterministic behavior are encoded into the models. This supports provably correct test model construction to perform test activities by avoiding controllability and observability issues.

- *Model Transformation*: We demonstrated how distributed communicating tester models can be generated automatically from centralized remote tester model (that is synthesized using existing methods) using novel algorithms and setup across geographical locations. We showed that the proposed architecture not only preserves the correctness of the testers but also mitigates the testing time, i.e., the distributed testers meets (one) Δ controllability requirement against 2Δ of the centralized remote tester.

- *Test Generation & Execution*: We presented the proposed test architecture and discussed its components such as test adapter between SUT and model adapter, coordination adapters to synchronize between local testers. We showed local testers capability to perform using coordination messages and trigger local test sequence at their ports. We presented test scenario where a centralized testing cannot be applied. The experiments also show that the distributed test architecture is more scalable and efficient in terms of test reaction time than centralized remote test architecture for testing large number of geographical locations (ports) in a system. We demonstrated the proposed test architecture using model-based testing platform DTRON as a suitable test execution platform for facilitating distributed testers deployment and management wherein the timing of coordination messages is implemented based on DTRON involved Spread tool.

- *Applicability of Approach*: Finally, feasibility of approach is demonstrated on industrial case study "Flexibility Contracts for Ancillary Services in Energy Grids". We showed how our model-based testing approach can be applied in industry for functional regression testing as well as for non-functional testing. To the best of author's knowledge, there is no such test automation method and non-commercial model-based testing tool yet to perform regression testing of time critical distributed systems.

5.1 Future Work

The research presented in thesis has shown a coherent approach for online model-based testing of real-time distributed systems. There are several potential directions of improvement for the approach and development of tool support for academic as well as for industrial need. We acknowledge that the findings of given approach have been derived from relatively few complex industrial case studies which are closed for public access and experimentation. To be able to generalize the experiment results, our approach would need to be applied on more examples with wide variety of features and scalability requirements. Due to constraints imposed by academic resources, developing a real-time system and testing it thoroughly, lies beyond the capacity and scope of this thesis.

Further validation and improvement of the results based on complex industrial case studies also requires the interests of industrial partners to collaborate actively. In the context of testing real-time systems, there are few high-level industries that develop real-time systems without restrictions on public access to their documentation and its implementation. We understand it is not easy to achieve this access and collaboration with industries due to privacy and security concern. Another concern is that industry user does not have enough time to learn details of formal modeling and the use of multiple supporting tools in the testing process. To bridge this gap we plan integrating the experimental tools used in this work to provide fully automated testing support that requires from user only knowing `SUT` and test purpose description in relevant domain-specific language. By improving the usability we expect attracting more industry applications to get empirical data on different scale use contexts, and usability features.

List of Figures

List of Tables

References

[1] Muthukumar, N.; Srinivasan, Seshadhri; Ramkumar, K.; Pal, Deepak; Vain, Jüri; Ramaswamy, Srini (2019). A model-based approach for design and verification of Industrial Internet of Things. Future Generation Computer Systems, 354–363.

[2] Pal, Deepak; Vain, Jüri (2019). A systematic approach on model refinement and regression testing of real-time distributed systems. 9th IFAC Conference on Manufacturing Modelling, Management and Control, MIM 2019 : Berlin, Germany, 28-30 August 2019, Proceedings. Ed. Ivanov, Dmitry; Dolgui, Alexandre; Yalaoui, Farouk. Elsevier, 1091-1096.

[3] Pal, Deepak; Vain, Jüri (2019). Model based test framework for communications-critical internet of things systems. Databases and Information Systems X: Selected Papers from the Thirteenth International Baltic Conference, DB&IS 2018. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Amsterdam: IOS Press, 79-94.

[4] Pal, Deepak; Vain, Jüri (2018). Model based approach for testing: distributed real-time systems augmented with online monitors. Databases and Information Systems : 13th International Baltic Conference, DB&IS 2018, Trakai, Lithuania, July 1-4, 2018, Proceedings. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Cham: Springer, 142-157.

[5] Pal, Deepak; Vain Jüri; Srinivasan, Seshadhri; Ramaswamy, Srini (2017). Model-based maintenance scheduling in flexible modular automation systems. 22nd IEEE International Conference on Emerging Technologies and Factory Automation, EFTA' 2017 : September 12-15, 2017, Limassol, Cyprus, 1-6.

[6] Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Automatic Distribution of Local Testers for Testing Distributed Systems. In: Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. (Ed.). Databases and Information Systems ix: Selected Papers from the Twelfth International Baltic Conference, DB&IS 2016 (297-310). Amsterdam: IOS Press. (Frontiers in Artificial Intelligence and Applications; 291).

[7] Pal, D.; Vain, J. (2016). Generating optimal test cases for real-time systems using DIVINE model checker. BEC 2016 : 15th Biennial Baltic Electronics Conference, Tallinn University of Technology, October 3-5, 2016 Tallinn, Estonia, 99–102.

[8] Anier, A.; Vain, J.; Tsiopoulos, L. (2017). DTRON: A tool for distributed model-based testing of time critical applications. Proceedings of the Estonian Academy of Sciences, 66 (1), 75-88.10.3176/proc.2017.1.08.

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

[10] Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis. Modeling for Verification, pages 75–105. Springer International Publishing, Cham, 2018.],[Formal Methods: An Appetizer Hardcover – July 17, 2019 by Flemming Nielson (Author), Hanne Riis Nielson (Author), Springer

[11] https://support.smartbear.com/testcomplete/docs/testing-with/advanced/distributed/basic-concepts.html

[12] M. Utting, B. Legeard. Practical Model-Based Testing. *Morgan Kaufmann, San Francisco*, 95,https://doi.org/10.1016/B978-012372501-1/50000-4, http://www.sciencedirect.com/science/article/pii/B9780123725011500004, 2007.

[13] Krichen, M., Tripakis, S.: Black-box conformance testing for real-time systems. In: 11th International SPIN Workshop on Model Checking Software. Volume 2989 of Lecture Notes in Computer Science., Springer (2004) 109–126

[14] Mikucionis, M., Larsen, K.G., Nielsen, B.: T-uppaal: Online model-based testing of realtime systems. In: 19th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (2004) 396–397

[15] Vain, J., Kaaramees, M., Markvardt, M.: *Online testing of nondeterministic systems with reactive planning tester*, In: Petre, L., Sere, K., Troubitsyna, E. (eds.) Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 113–150. IGI Global, Hershey, 2012.

[16] R. Alur and D. Dill. A theory of timed automata. Theoretical Computer Science B, 126:183–235, 1994.

[17] Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools, In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets, vol. 3098, pp. 87–124. (2004)

[18] K.G. Larsen, P. Pettersson, and Y. Wang, Uppaal in a Nutshell, Intel J. Software Tools for Technology Transfer, vol. 1, nos. 1-2, pp. 134-152, 1997

[19] Behrmann, G., David, A., Larsen, K. G.: *A Tutorial on Uppaal*, In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg, 2004.

[20] K.Sarna , J. Vain. Exploiting aspects in model-based testing. FOAL12: Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages pp. 45-47, 2012

[21] Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using uppaal. In: Formal methods and testing. Springer (2008) 77–117

[22] Segala, R.: *Quiescence, fairness, testing, and the notion of implementation*, In: Best, E. (eds.) 4th Intrenational Conference on Concurrency Theory (CONCUR'93). LNCS, vol. 715, pp. 324–338. Springer, Heidelberg 1993.

[23] G. Luo, R. Dssouli, G.v. Bochmann, P. Venkataram, and A.Ghedamsi, "Test Generation with Respect to Distributed Interfaces," Computer Standards and Interfaces, vol. 16, pp. 119–132, 1994

[24] L. Cacciari and O. Rafiq, "Controllability and Observability in Distributed Testing," Information and Software Technology, 1999.

[25] M. Benattou, L. Cacciari, R. Pasini, and O. Rafiq, "Principles an Tools for Testing Open Distributed Systems," Proc. 12th Int'l Workshop Testing of Communicating Systems (IWTCS), pp. 77–92, Sept. 1999.

[26] A. Khoumsi, "Timing Issues in Testing Distributed Systems," Proc. Fourth IASTED Int'l Conf. Software Eng. Applications (SEA), Nov. 2000.

[27] Segala, R.: *Quiescence, fairness, testing, and the notion of implementation*, In: Best, E. (eds.) 4th Intrenational Conference on Concurrency Theory (CONCUR'93). LNCS, vol. 715, pp. 324–338. Springer, Heidelberg 1993.

[28] S. Seshadhri, B. Furio, Vain J., Ramaswamy S., *Model checking response times in Networked Automation Systems using jitter bounds*, Computers in Industry, vol. 74, pp. 186–200, 2015, Elsevier.

[29] S. Seshadhri, B. Furio, Ramaswamy S, Vain J., *Verifying response times in networked automation systems using jitter bounds*, Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, pp. 47–50, 2014, IEEE.

[30] Balasubramaniyan S. S. Seshadhri, B. Furio, Vain J, Ramaswamy S., *Design and verification of Cyber-Physical Systems using TrueTime, evolutionary optimization and UPPAAL*, Microprocessors and Microsystems, vol. 42, pp. 37–48, 2016, Elsevier.

[31] T. S. Chow, "Testing software design modelled by finite state machines," IEEE Trans. Softw. Eng., vol. SE-4, no. 3, pp. 178–187, May 1978.

[32] J. Tretmans, "Model based testing with labelled transition systems," in Formal Methods and Testing, New York, NY, USA: Springer, 2008, pp. 1–38.

[33] *ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework* - Parts 1-5. International Standard IS-9646. ISO, Geneve, 1991.

[34] Sarikaya, B., v. Bochmann, G.: *Synchronization and specification issues in protocol testing*, In: IEEE Trans. Commun., pp. 389–395. IEEE Press, New York (1984)

[35] M. Benattou, L. Cacciari, R. Pasini, and O. Rafiq, *Principles an Tools for Testing Open Distributed Systems*, Proc. 12th International Workshop Testing of Communicating Systems (IWTCS), pp. 77–92, Sept. 1999.

[36] L. Cacciari and O. Rafiq, *Controllability and Observability in Distributed Testing*, Information and Software Technology, 1999.

[37] Hierons, R. M., Merayo, M. G., Nunez, M.: *Implementation relations and test generation for systems with distributed interfaces*, Distributed Computing, vol. 25, no. 1, pp. 35–62. Springer-Verlag, 2012.

[38] David, A., Larsen, K. G., Mikuionis, M., Nguena Timo, O. L., Rollet, *A: Remote Testing of Timed Specifications*, In: Proceedings of the 25th IFIP International Conference on Testing Software and Systems (ICTSS 2013), pp. 65–81. Springer, Heidelberg ,2013.

[39] Vain, J., Kaaramees, M., Markvardt, M.: *Online testing of nondeterministic systems with reactive planning tester*, In: Petre, L., Sere, K., Troubitsyna, E. (eds.) Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 113–150. IGI Global, Hershey, 2012.

[40] Khoumsi Ahmed, *A Temporal Approach for Testing Distributed Systems*, Journal IEEE Transactions on Software Engineering, Volume 28 Issue 11, Page 1085-1103, IEEE Press Piscataway, NJ, USA, November 2002.

[41] Wachter, B., Genon, A., Massart, T., Meuter, C.: The formal design of distributed controllers with dSL and Spin, Formal Aspects of Computing", pp. 177–200. (2005)

[42] Cimatti, A., Clarke, M., Enrico, G., Fausto, G., Marco, P., and Armando, T.: Nusmv 2: An Open Source Tool for Symbolic Model Checking, In: CAV, (2002)

[43] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, vol. 1032 Springer, (1996)

[44] Clarke, E., Grumberg, O., and Peled. D.: Model Checking. The MIT Press, (1999)

[45] McMillan, K. L.: Symbolic model checking: an approach to the state explosion problem. Carnegie Mellon University, (1992)

[46] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing," 2006.

[47] M. Timmer, H. Brinksma, and M. Stoelinga, Model-based testing, 2011.

[48] R. Alur and D. Dill. A theory of timed automata. Theoretical Computer Science, 126:183–235, 1994.

[49] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In Compositionality, volume 1536 of LNCS. Springer, 1998.

[50] J. Sifakis and S. Yovine. Compositional specification of timed systems. In 13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96, volume 1046 of LNCS. Spinger-Verlag, 1996.

[51] Goodloe, A., Pike, L.: Monitoring distributed real-time systems: a survey and future directions (NASA/CR-2010-216724), Havelund, K., Rosu, G.: Synthesizing monitors for safety properties, (2010)

[52] A. Bauer, M. Leucker and C. Schallhart: Model-based runtime analysis of distributed reactive systems, Australian Software Engineering Conference, pp. 10. (2006)

[53] K. Sen, A. Vardhan, G. Agha and G. Rosu: Efficient decentralized monitoring of safety in distributed systems, In: Proceedings of 26th International Conference on Software Engineering, pp. 418-427. (2004)

[54] A. Chatterjee, A. Bala, M. Shah and A. H. Nagappa, "CTAF: Centralized Test Automation Framework for multiple remote devices using XMPP," 2018 15th IEEE India Council International Conference (INDICON), Coimbatore, India, 2018, pp. 1-6, doi: 10.1109/INDICON45594.2018.8987182.

[55] Sandeep K.S. Gupta, Tridib Mukherjee, Georgios Varsamopoulos, and Ayan Banerjee. Research directions in energy-sustainable cyber-physical systems. Sustainable Computing: Informatics and Systems, 1(1):57 – 74, 2011.

[56] Chapter 1 - the challenge. In Mark Utting and Bruno Legeard, editors, Practical Model-Based Testing, pages 1 – 18. Morgan Kaufmann, San Francisco, 2007. Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of modelbased testing approaches. Software testing, verication and reliability, 22(5):297– 312, 2012.

[57] Robert V Binder, Bruno Legeard, and Anne Kramer. Model-based testing: Where does it stand? Queue, 13(1):40–48, 2014.

[58] Hierons, Robert M, Merayo Mercedes G, Núñez Manuel Controllability Through Nondeterminism in Distributed Testing. Testing Software and Systems, 2016, Springer International Publishing, 89–105, isbn=978-3-319-47443-4.

[59] A Systematic Review of Model Based Testing Tool Support Muhammad Shafique, Yvan Labiche.

[60] https://en.wikipedia.org/wiki/Boeing_737_MAX_groundings

[61] https://blogs.nasa.gov/commercialcrew/2020/02/07/nasa-shares-initial-findings-from-boeing-starliner-orbital-flight-test-investigation/

[62] Brook, A.: Evolution and Practice: Low-latency Distributed Applications in Finance. Queue - Distributed Computing, vol. 13, no. 4, pp. 40–53. ACM, New York (2015).

[63] Reinhart Richter, Xcerra Corporation, Does the Internet of Things force us to rethink our test strategies? http://xcerra.com/ep doestheinternetofthingsforceustorethink-ourteststrategiesvision

[64] Vain, J., Kääramees, M., Markvardt, M.: Online testing of nondeterministic systems with reactive planning tester. In: Petre, L., Sere, K., Troubitsyna, E. (eds.) Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 113–150. IGI Global, Hershey (2012)

[65] Hierons, M., Merayo, G., Núñez, Manuel: Timed implementation relations for the distributed test architecture, Distributed Computing, Vol. 27, (2014)

[66] S. Gulwani, O. Polozov and R. Singh. (2017). Program synthesis. In series Foundations and Trends® inProgramming Languages. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/10/programsynthesisnow.pdf

[67] Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, p. 201-216

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Jüri Vain, for trusting in me, for his constant encouragement, guidance, patience and support through all the years. It has certainly been an honor to work with him. Prof. Vain gave me much freedom to explore interesting problems of software testing areas, while providing invaluable insights and comments on my thoughts. My research has been benefited greatly from breadth and depth of his knowledge. His high standards played a key role to build up my confidence in both the normal and research life, which is the most precious achievement I am gaining working with him.

Secondly, I would like to sincerely thank Prof. Johan Lilius and Prof. Artem Boyarchuk for their time and effort to review my thesis. Their valuable feedback and comments are greatly appreciated. I am also honored and thankful to both Prof. Johan and and Prof. Artem for their kind acceptance to act as the opponent at my doctoral defence.

Finally, I want to thank my wife Apneet for the support, encouragement and patience. I am grateful for all your support and love through all the years.

Abstract
Model-Based Testing of Real-Time Distributed Systems

Real-time computer-based systems are woven into the fabric of our lives. They are embedded in distributed cyber-physical systems such as critical infrastructures, manufacturing systems, traffic control and many other which involve numerous interacting heterogeneous components from micro to macro scale. Due to the criticality and complexity of such systems their development process presumes proper support by software quality assurance methods and tools.

In this thesis, we present a model-based testing method, test architecture and provably correct test development work-flow for time critical distributed systems. The approach aims at *online* model-based testing of distributed systems with the focus on test automation and provable test quality. Another focus of this research is tests timing and performance aspects that when ignored cause dropping the usability of testing tools especially for complex distributed systems.

To address these issues and better meet the demand for advanced testing approaches, we propose a testing process pipeline starting from modelling distributed systems as n-port Uppaal Timed Automata, and finishing with test deployment algorithms to distribute the monolithic test models between local test execution agents. The deployment algorithms assure correct synchronization and coordination of local tests over the system under test. As a main result, it is shown that due to proposed test deployment algorithms the distributed test architecture not only preserves the correctness of the test models generated for centralized remote testing but it also mitigates the testing time. While the best known result of centralized remote testing approaches provides 2Δ-controllability of tests our distributed testing architecture meets (one) Δ-controllability requirement where Δ denotes the communication delay upper bound between testers and/or system under test.

The feasibility of our approach is demonstrated on industrial case study "*Flexibility Contracts for Ancillary Services in Energy Grids*". We show how our model-based testing approach can be applied in industry for functional regression testing as well as for non-functional testing. To the best of author's knowledge, there is no such test automation method and non-commercial model-based testing tool yet to perform regression testing of time critical distributed systems.

Kokkuvõte
Reaalaja hajussüsteemide mudelipõhine testimine

Reaalaja arvutisüsteemid on lahutamatu osa meie tehnoloogilisest keskkonnast. Neile baseeruvad küberfüüsikalised süsteemid nagu näiteks kriitilised infrastruktuurid, tootmissüsteemid, liikluse reguleerimisesüsteemid ja paljud teised, kus suur hulk heterogeenseid komponente mikrost- makrotasemeni on omavahel pidevas interaktsioonis. Oma keerukuse ja rakenduskriitilisuse tõttu eeldab niisuguste süsteemide arendusprotsess olulisel määral tarkvara kvaliteedi tagamise meetodite ja neid toetavate tööriistade kasutamist.

Käesoleva väitekirja raames on loodud mudelipõhine testimismeetod, testiarhitektuur ja tõestatavalt korrektsete testide arenduse töövoog, mis on orienteeritud ajakriitiliste hajussüsteemide testimisele. Loodud lähenemine keskendub hajussüsteemide *online* testimisele ning selle automatiseerimisele eesmärgiga tagada testide ja testitulemuste tõestuspõhine kvaliteet. Uuringu teine põhifookus puudutab testide ajastuskorrektsuse ja jõudluse aspekte, mille ignoreerimine võib viia testimisvahendite kasutatavuse olulise vähenemiseni, seda eriti kompleksete hajussüsteemide korral.

Pakkumaks uudset lähenemist eelmainitud probleemide lahendamiseks, esitatakse väitekirjas testimisprotsessi töövoog, mis katab samme alates hajussüsteemide modelleerimisest n-pordiga Uppaali ajaga automatide mudeliga, lõpetades genereeritud monoliitse testimudeli jaotamisega lokaalseid teste täitvate süsteemi osade vahel. Monoliitse testimudeli jaotamisalgoritmid lisavad lokaalsetele testikomponentidele sünkroniseerimis- ja koordineerimismehhanismi, mis tagab mitte ainult hajustestide korrektsuse säilimise nende tuletamisel tsentraliseeritud kaugtestimise mudelitest, vaid parandab ka testide reaktsiooniaega. Kui teadaolevalt parimad tsentraliseeritud kaugtestimise meetodid tagavad testide reaktsiooniaja 2Δ, nn 2Δ- juhitavuse, siis väitekirjas loodud hajustesti arhitektuur rahuldab (ühe) Δ- juhitavuse nõuet, kus Δ tähistab testrite omavahelisest või testri ja süsteemi vahelisest kommunikatsioonist tingitud hilistumise ülemist raja.

Väitekirjas loodud lahenduste otstarbekust demonstreeritakse tööstuslikul rakendusnäitel "*Flexibility Contracts for Ancillary Services in Energy Grids*", kus energiajaotuse teenuste integratsiooni taseme testimiseks genereeritakse testimudelid ja verifitseeritakse nende korrektsus. Mudeleid on rakendatud teenuste nii funktsionaalsete- kui ka mittefunktsionaalsete omaduste testimisel. Väitekirja autorile teadaolevalt ei ole varem antud lähenemist rakendatud ajakriitiliste teenuste integratsioonitestimisel. Samuti puuduvad mittekommertsiaalsed mudelipõhise testimise vahendid keerukate ajakriitiliste hajussüsteemide regressioontestimiseks.

Appendix 1

I

Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Automatic Distribution of Local Testers for Testing Distributed Systems. In: Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. (Ed.). Databases and Information Systems ix: Selected Papers from the Twelfth International Baltic Conference, DB&IS 2016 (297-310). Amsterdam: IOS Press. (Frontiers in Artificial Intelligence and Applications; 291)

Databases and Information Systems IX
G. Arnicans et al. (Eds.)

doi:10.3233/978-1-61499-714-6-297

Automatic Distribution of Local Testers for Testing Distributed Systems

Jüri VAIN [1], Evelin HALLING, Gert KANTER, Aivo ANIER and Deepak PAL
Department of Computer Science, Tallinn University of Technology, Estonia
http://cs.ttu.ee/

Abstract. *Low-latency systems* where reaction time is primary success factor and design consideration, are serious challenge to existing integration and system level testing techniques. Modern cyber physical systems have grown to the scale of global geographic distribution and latency requirements are measured in nanoseconds. While existing tools support prescribed input profiles they seldom provide enough reactivity to run the tests with simultaneous and interdependent input profiles at remote front ends. Additional complexities emerge due to severe timing constraints the tests have to meet when test navigation decision time ranges near the message propagation time. Sufficient timing conditions for remote online testing have been proposed in remote Δ-testing method recently. We extend the Δ-testing by deploying testers on fully distributed test architecture. This approach reduces the test reaction time by almost a factor of two. We validate the method on a distributed oil pumping SCADA system case study.

Keywords. model-based testing, distributed systems, low-latency systems

1. Introduction

Modern large scale cyber-physical systems have grown to the size of global geographic distribution and their latency requirements are measured in microseconds or even nanoseconds. Such applications where latency is one of the primary design considerations are called *low-latency systems* and where it is of critical importance – to *time critical systems*. A typical example of distributed time critical system is smart energy grid (SEG) where delayed control signals can cause overloads and blackouts of whole regions. Thus, the proper timing is the main measure of success in SEG and often the hardest design concern.

Since large SEG-s systems are mostly distributed systems (by distributed systems we mean the systems where computations are performed on multiple networked computers that communicate and coordinate their actions by passing messages), their latency dynamics is influenced by many technical and non-technical factors. Just to name a few, energy consumption profile look up time (few milliseconds) may depend on the load profile, messaging middleware and the networking stacks of operating systems. Similarly, due to cache miss, the caching time can grow from microseconds to about hundred

[1]Corresponding Author: Jüri Vain; Department of Computer Science, Tallinn University of Technology, Akadeemia tee 15A, 19086 Tallinn, Estonia; E-mail: juri.vain@ttu.ee

milliseconds [1]. Reaching sufficient feature coverage by integration testing of such systems in the presence of numerous latency factors and their interdependences, is out of the reach of manual testing. Obvious implication is that scalable integration and system level testing presumes complex tools and techniques to assure the quality of the test results [2]. To achieve the confidence and trustability, the test suites need to be either correct by construction or verified against the test goals after they are generated. The need for automated test generation and their correctness assurance have given raise to model based testing (MBT) and the development of several commercial and academic MBT tools. In this paper, we interpret MBT in the standard way, i.e. as conformance testing that compares the expected behaviors described by the system requirements model with the observed behaviors of an actual implementation (implementation under test). For detailed overview of MBT and related tools we refer to [3] and [4].

2. Related Work

Testing distributed systems has been one of the MBT challenges since the beginning of the 90s. An attempt to standardize the test interfaces for distributed testing was made in ISO OSI Conformance Testing Methodology [5]. A general distributed test architecture, containing distributed interfaces, has been presented in Open Distributed Processing (ODP) Basic Reference Model (BRM), which is a generalized version of ISO distributed test architecture. First MBT approaches represented the test configurations as systems that can be modeled by finite state machines (FSM) with several distributed interfaces, called ports. An example of abstract distributed test architecture is proposed in [6]. This architecture suggests the Implementation Under Test (IUT) contains several ports that can be located physically far from each other. The testers are located in these nodes that have direct access to ports. There are also two strongly limiting assumptions: (i) the testers cannot communicate and synchronize with one another unless they communicate through the IUT, and (ii) no global clock is available. Under these assumptions a test generation method was developed in [6] for generating synchronizable test sequences of multi-port finite state machines. However, it was shown in [7] that no method that is based on the concept of synchronizable test sequences can ensure full fault coverage for all the testers. The reason is that for certain testers, given a FSM transition, there may not exist any synchronizable test sequence that can force the machine to traverse this transition. This is generally known as *controllability* and *observability* problem of distributed testers. These problems occur if a tester cannot determine either when to apply a particular input to IUT, or whether a particular output from IUT is generated in response to a specific input [8]. For instance, the controllability problem occurs when the tester at a port p_i is expected to send an input to IUT after IUT has responded to an input from the tester at some other port p_j, without sending an output to p_i. The tester at p_i is unable to decide whether IUT has received that input and so cannot know when to send its input. Similarly, the observability problem occurs when the tester at some port p_i is expected to receive an output from IUT in response to a given input at some port other than p_i and is unable to determine when to start and stop waiting. Such observability problems can introduce fault masking.

In [8], it is proposed to construct test sequences that cause no controllability and observability problems during their application. Unfortunately, offline generation of

test sequences is not always applicable. For instance, when the model of IUT is non-deterministic it needs instead of fixed test sequences online testers capable of handling non-deterministic behavior of IUT. But even this is not always possible. An alternative is to construct testers that includes external coordination messages. However, that creates communication overhead and possibly the delay introduced by the sending of each message. Finding an acceptable amount of coordination messages depends on timing constraints and finally amounts to finding a tradeoff between the controllability, observability and the cost of sending external coordination messages.

The need for retaining the timing and latency properties of testers became crucial natively when time critical cyber physical and low-latency systems were tested. Pioneering theoretical results have been published on test timing correctness in [9] where a remote abstract tester was proposed for testing distributed systems in a centralized manner. It was proven that if IUT ports are remotely observable and controllable then 2Δ-condition is sufficient for satisfying timing correctness of the test. Here, Δ denotes an upper bound of message propagation delay between tester and IUT ports. However, this condition makes remote testing problematic when 2Δ is close to timing constraints of IUT, e.g. the length of time interval when the test input has to reach port has definite effect on IUT. If the actual time interval between receiving an IUT output and sending subsequent test stimulus is longer than 2Δ the input may not reach the input port in time and the test goal cannot be reached.

In this paper we focus on distributed online testing of low latency and time-critical systems with distributed testers that can exchange synchronization messages that meet Δ-delay condition. In contrast to the centralized testing approach, our approach reduces the tester reaction time from 2Δ to Δ. The validation of proposed approach is demonstrated on a distributed oil pumping SCADA system case study.

3. Preliminaries

3.1. Model-Based Testing

In model-based testing, the formal requirements model of implementation under test describes how the system under test is required to behave. The model, built in a suitable machine interpretable formalism, can be used to automatically generate the test cases, either offline or online, and can also be used as the oracle that checks if the IUT behavior conforms to this model. Offline test generation means that tests are generated before test execution and executed when needed. In the case of online test generation the model is executed in lock step with the IUT. The communication between the model and the IUT involves controllable inputs of the IUT and observable outputs of the IUT.

There are multiple different formalisms used for building conformance testing models. Our choice is Uppaal timed automata (TA) [10] because the formalism is designed to express the timed behavior of state transition systems and there exists a family of tools that support model construction, verification and online model-based testing [11].

3.2. Uppaal Timed Automata

Uppaal Timed Automata [10] (UTA) used for the specification of the requirements are defined as a closed network of extended timed automata that are called *processes*. The

processes are combined into a single system by the parallel composition known from the process algebra CCS. An example of a system of two automata comprised of 3 locations and 2 transitions each is given in Figure 1.

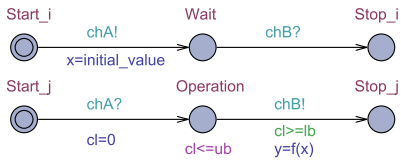


Figure 1. A parallel composition of Uppaal timed automata

The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by an additional circle inside the location.

Synchronous communication between the processes is by hand-shake synchronization links that are called *channels*. A channel relates a pair of edges labeled with symbols for input actions denoted by e.g. `chA?` and `chB?` in Figure 1, and output actions denoted by `chA!` and `chB!`, where `chA` and `chB` are the names of the channels.

In Figure 1, there is an example of a model that represents a synchronous remote procedure call. The calling process Process_i and the callee process Process_j both include three locations and two synchronized transitions. Process_i, initially at location Start_i, initiates the call by executing the send action `chA!` that is synchronized with the receive action `chA?` in Process_j, that is initially at location Start_j. The location Operation denotes the situation where Process_j computes the output `y`. Once done, the control is returned to Process_i by the action `chB!`

The duration of the execution of the result is specified by the interval $[lb, ub]$ where the upper bound *ub* is given by the *invariant* `cl<=ub`, and the lower bound *lb* by the *guard condition* `cl>=lb` of the transition Operation $\rightarrow$ Stop_j. The *assignment* `cl=0` on the transition Start_j $\rightarrow$ Operation ensures that the clock *cl* is reset when the control reaches the location Operation. The global variables `x` and `y` model the input and output arguments of the remote procedure call, and the computation itself is modelled by the function `f(x)` defined in the declarations section of the Uppaal model.

The inputs and outputs of the test system are modeled using channels labeled in a special way described later. Asynchronous communication between processes is modeled using global variables accessible to all processes.

Formally the Uppaal timed automata are defined as follows. Let Σ denote a finite alphabet of actions $a, b, \ldots$ and C a finite set of real-valued variables p, q, r, denoting clocks. A guard is a conjunctive formula of atomic constraints of the form $p \sim n$ for $p \in C, \sim \in \{\geq, \leq, =, >, <\}$ and $n \in \mathbb{N}^+$. We use $G(C)$ to denote the set of clock guards. A timed automaton A is a tuple $\langle N, l_0, E, I \rangle$ where N is a finite set of locations (graphically denoted by nodes), $l_0 \in N$ is the initial location, $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges (an edge is denoted by an arc) and $I : N \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $p \leq n$ or $p < n, n \in \mathbb{N}^+$. Without the loss of generality we assume that guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer variables. Similarly to

clock conditions, the propositions on integer variables k are of the form $k \sim n$ for $n \in \mathbb{N}$, and $\sim \in \{\leq, \geq, =, >, <\}$. For the formal definition of Uppaal TA full semantics we refer the reader to [12] and [10].

4. Remote Testing

The test purpose most often used in MBT is conformance testing. In conformance testing the IUT is considered as a black-box, i.e., only the inputs and outputs of the system are externally controllable and observable respectively. The aim of black-box conformance testing according to [13] is to check if the behavior observable on system interface conforms to a given requirements specification. During testing, a tester executes selected test cases on an IUT and emits a test verdict (pass, fail, inconclusive). The verdict shows correctness in the sense of input-output conformance relation (IOCO) between IUT and the specification. The behavior of a IOCO-correct implementation should respect after some observations following restrictions:

(i) the outputs produced by IUT should be the same as allowed in the specification;

(ii) if a quiescent state (a situation where the system can not evolve without an input from the environment [14]) is reached in IUT, this should also be the case in the specification;

(iii) any time an input is possible in the specification, this should also be the case in the implementation.

The set of tests that forms a test suite is structured into test cases, each addressing some specific test purpose. In MBT, the test cases are generated from formal models that specify the expected behavior of the IUT and from the coverage criteria that constrain the behavior defined in IUT model with only those addressed by the test purpose. In our approach Uppaal Timed Automata (UTA) [10] are used as a formalism for modeling IUT behavior. This choice is motivated by the need to test the IUT with timing constraints so that the impact of propagation delays between the IUT and the tester can be taken into account when the test cases are generated and executed against remote real-time systems.

Another important aspect that needs to be addressed in remote testing is functional non-determinism of the IUT behavior with respect to test inputs. For nondeterministic systems only online testing (generating test stimuli on-the-fly) is applicable in contrast to that of deterministic systems where test sequences can be generated offline. Second source of non-determinism in remote testing of real-time systems is communication latency between the tester and the IUT that may lead to interleaving of inputs and outputs. This affects the generation of inputs for the IUT and the observation of outputs that may trigger a wrong test verdict. This problem has been described in [15], where the Δ-testability criterion (Δ describes the communication latency) has been proposed. The Δ-testability criterion ensures that wrong input/output interleaving never occurs.

4.1. Centralized Remote Testing

Let us first consider a centralized tester design case. In the case of centralized tester, all test inputs are generated by a single monolithic tester. This means that the centralized tester will generate an input for the IUT, waits for the result and continues with the next set of inputs and outputs until the test scenario has been finished. Thus, the tester has to

wait for the duration it takes the signal to be transmitted from the tester to the IUT's ports and the responses back from ports to the tester. In the case of IUT being distributed in a way that signal propagation time is non-negligible, this can lead into a situation where the tester is unable to generate the necessary input for the IUT in time due to message propagation latency. These timing issues can render testing an IUT impossible if the IUT is a distributed real-time system.

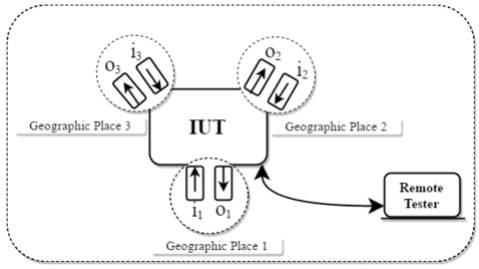


Figure 2. Remote tester communication architecture

To be more concrete, let us consider the remote testing architecture depicted in Figure 2 and the corresponding model depicted in Figure 3 and 4. In this case the IUT has 3 ports (p_1, p_2, p_3) in geographically different places to interact within the system, inputs i_1, i_2 and i_3 at ports p_1, p_2 and p_3 respectively and outputs o_1 at port p_1, o_2 at port p_2, o_3 at port p_3.

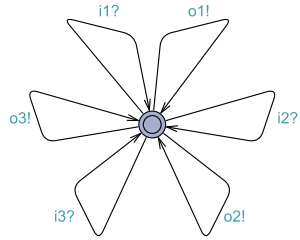


Figure 3. IUT model

We model a multi-ports timed automata by splitting the edges with multiple communication actions to a sequence of edges each labeled with exactly one action and connected via committed locations, so that all ports of such group are updated at the same time. In Figure 4 the labels on the edges represent the transitions and the transition tuple ($\texttt{L0}, \texttt{L1}, i_1!/(o_1?, o_2?)$) is represented by sequence of edges each labeled with exactly one action and connected via committed locations. For example the sequence of edges from location `L0` to `L1` with labels $i_1!$, $o_1?$ and $o_2?$ represents the multiple communication actions where the input $i_1!$ at port p_1 in location `L0` being able to trigger a transition that leads to the output $o_1?$ and $o_2?$ at ports p_1, p_2 respectively and the location becoming `L1`.

Using such splitting of edges with committed locations, we model a three port automata shown in Figure 4 where the tester sends an input i_1 to the port p_1 at `Geographic Place 1` and receives a response or outputs o_1 and o_2 from IUT at `Geographic Place 1` and `Geographic Place 2` respectively. After receiving the result, the tester is in location `L1`, it gets both i_3 on port p_3 and i_2 on port p_2. Then, either it follows the intended

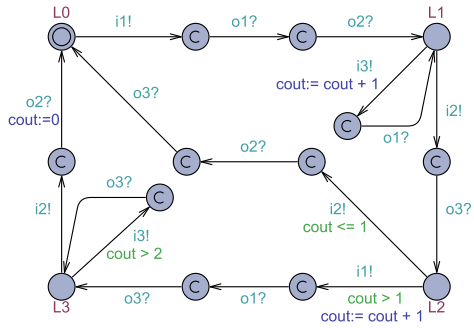


Figure 4. Remote Tester model

path sending i_3 before i_2, or it sends i_2 before i_3. If tester decides to send i_3 before i_2 it receives an output o_1 at port p_1 and returns to location `L1`. The transition is a self loop if its start and end locations are the same. If tester decides to send i_2, the IUT responds with an output o_3 at port p_3. Now, the tester is in location `L2`, it gets both i_1 on port p_1 and i_2 on port p_2. Based on guard condition and previously triggered inputs and received outputs the next input is sent to IUT and tester continues with the next set of inputs and outputs until the test scenario has been finished.

The described IUT is a real-time distributed system, which means that it has strict timing constraints for messaging between ports. More specifically, after sending the first input i_1 to port p_1 at `Geographic Place 1` and after receiving the response o_1 and o_2 at `Geographic Place 1` and `Geographic Place 2` respectively, the tester needs to decide and send the next input i_2 to port p_2 at `Geographic Place 2` or input i_3 to port p_3 at `Geographic Place 3` in Δ time. But, due to the fact that the tester is not at the same geographical place as the distributed IUT, it is unable to send the next input in time as the time it takes to receive the response and send the next input amounts to 2Δ, which is double the time allotted for the next input signal to arrive.

Consequently, the centralized remote testing approach is not suitable for testing a real-time distributed system if the system has strict timing constraints with non negligible signal propagation times between system ports. To overcome this problem, the centralized tester is decomposed and distributed as described in the next section.

5. Distributed Testing

The shortcoming of the centralized remote testing approach is mitigated with extending the Δ-testing idea by decomposing the monolithic remote tester into multiple local testers. These local testers are directly attached to the ports of the IUT. Thus, instead of bidirectional communication between a remote tester and the IUT, only unidirectional synchronization between the local testers is required. The local testers are generated in two steps: at first, a centralized remote tester is generated by applying the reactive planning online-tester synthesis method of [16], and second, a set of synchronizing local testers is derived by decomposing the monolithic tester into a set of location specific tester instances. Each tester instance needs to know now only the occurrence of i/o events

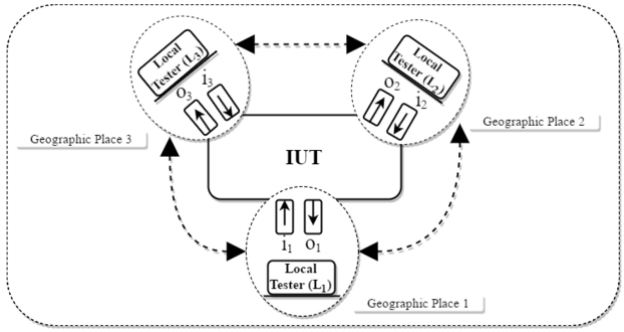


Figure 5. Distributed Local testers communication architecture

at other ports which determine its behavior. Possible reactions of the local tester to these events are already specified in its model and do not need further feedback to the event sender. The decomposing preserves the correctness of testers so that if the monolithic remote tester meets 2Δ requirement then the distributed testers meet (one) Δ-controllability requirement.

We apply the algorithm described in 5.1 to transform the centralized testing architecture depicted in Figure 2 into a set of communicating distributed local testers, the architecture of which is shown in Figure 5. After applying the algorithm, the message propagation time between the local tester and the IUT port has been eliminated because the tester is attached directly to the port. This means that the overall testing response time is also reduced, because previously the messages had to be transmitted over a channel with latency bidirectionally. The resulting architecture mitigates the timing issue by replacing the bidirectional communication with a unidirectional broadcast of the IUT output signals between the distributed local testers. The generated local tester models are shown in Figure 6, Figure 7, Figure 8 and Figure 9.

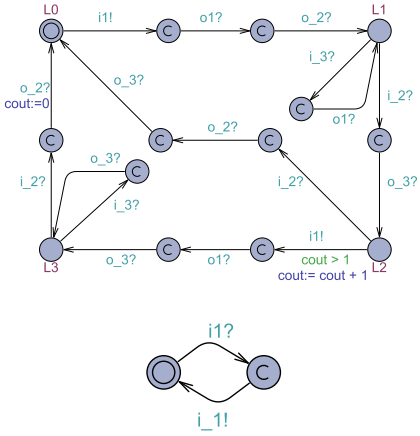


Figure 6. Local tester at Geographic Place 1

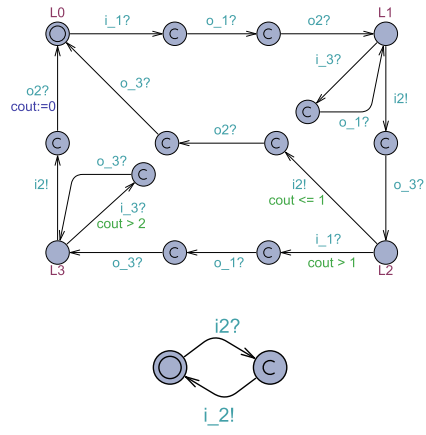


Figure 7. Local tester at Geographic Place 2

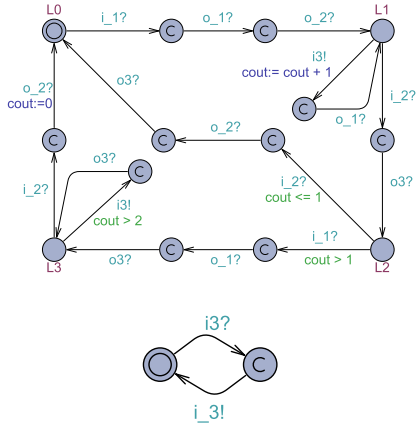


Figure 8. Local tester at Geographic Place 3

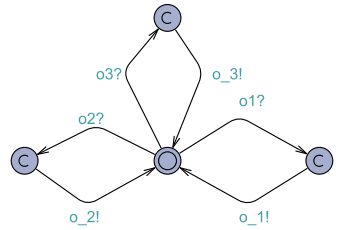


Figure 9. Output Event Synchronizer

5.1. Tester Distribution Algorithm

Let M^{MT} denote a monolithic remote tester model generated by applying the reactive planning online-tester synthesis method [16]. $Loc(IUT)$ denotes a set of geographically different port locations of IUT. The number of locations can be from 1 to n, where $n \in \mathbb{N}$ i.e. $Loc(IUT) = \{l_n | n \in \mathbb{N}\}$. Let P_{l_n} denotes a set of ports accessible in the location l_n.

1. For each l, $l \in Loc(IUT)$ we copy M^{MT} to M^l to be transformed to a location specific local tester instance.
2. For each M^l we go through all the edges in M^l. If the edge has a synchronizing channel and the channel does not belong to the the set of ports P_{l_n}, we do the following:
   - if the channel's action is *send*, we replace it with the co-action *receive*.
   - if the channel's action is *receive*, we do nothing.
3. For each M^l we add one more automaton that duplicates the input signals from M^l to IUT, attached to the set of ports P_{l_n} and broadcasts the duplicates to other local testers to synchronize the test runs at their local ports. Similarly the IUT local output event observations are broadcast to other testers for synchronization purposes like automaton in Figure 9.

6. Correctness of Tester Distribution Algorithm

To verify the correctness of distributed tester generation algorithm we check the bi-simulation equivalence relation between the model of monolithic centralized tester and that of distributed tester. For that the models are composed by parallel compositions so that one has a role of words generator on i/o alphabet and other the role of words acceptor machine. If the i/o language acceptance is established in one direction then the roles of models are reversed. Since the i/o alphabets of remote tester and distributed tester differ due to synchronizing messages of distributed tester the behaviors are compared based on the i/o alphabet observable on IUT ports only. Second adjustment of models to be made

for bi-simulation analysis is the reduction of message propagation delays to uniform basis either on Δ or 2Δ in both models. Assume (due to closed world assumption used in MBT):

- the centralized remote tester model: $M^{remote} = TA^{IUT} \parallel TA^{r-TST}$
- the distributed tester model: $M^{distrib} = TA^{IUT} \parallel_{||_i} TA_i^{d-TST}$ $i = [1, n]$, n - number of ports locations.
- to unify the timed words $TW(M^{remote})$ and $TW(M^{distrib})$ the communication delay between IUT and Tester is assumed.

Definition (correctness of tester distribution mapping): The mapping $M^{remote} \xrightarrow{Algorithm} M^{distrib}$ is correct if TA^{r-TST} and $||_i TA_i^{d-TST}$ are observation bisimilar, i.e. if TA^{r-TST} and $||_i TA_i^{d-TST}$ are respectively generating and accepting automata on common i/o alphabet $\Sigma^i \cup \Sigma^o$ then all timed words $TW(TA^{r-TST})$ are recognizable by $||_i TA_i^{d-TST}$ and all timed words $TW(||_i TA_i^{d-TST})$ are recognizable by TA^{r-TST}.

Here, alphabet $\Sigma^i \cup \Sigma^o$ includes i/o symbols used at IUT-TESTER interfaces of M^{remote} and $M^{distrib}$.

Correctness verification of the distribution mapping:

Step 1: (Constructing generating-accepting automata synchronous composition):

- label each output action of TA^{r-TST} with output symbol a! and its co-action in $||_i TA_i^{d-TST}$ with input symbol a?;
- define parallel composition $TA^{r-TST} \parallel_{||_i} TA_i^{d-TST}$ with synchronous i/o actions.

Step 2: (Bisimilarity proof by model checking): TA^{r-TST} and $||_i TA_i^{d-TST}$ are observation bisimilar if following holds: $M^{remote} \models \texttt{not deadlock} \wedge M^{distrib} \models \texttt{not deadlock} \Rightarrow TA^{r-TST} \parallel_{||_j} TA_j^{d-TST} \models \texttt{not deadlock}$ $j = [1, n]$, n - number of local testers , i.e. the composition of bisimilar testers must be non-blocking if the testers composed with IUT model separately are non-blocking.

7. Case Study

7.1. Use Case

The benefit of using the proposed method is demonstrated in the use case of an EMS (Energy Management System) which is integrated into the SCADA (Supervisory Control And Data Acquisition) system of an industrial consumer. An EMS is essentially a load balancing system. The target of the balancing system is the load on power supplies called feeders to an industrial consumer. These industrial power consumers have multiple feeders to power the devices required for their operations (e.g., pumps and pipeline heating systems). The motivation for balancing the power consumption between the feeders stems from the fact that the power companies can enforce fines on the industrial consumers if the power consumption exceeds certain thresholds due to safety considerations and possible damage to the equipment. Therefore, the consumer is motivated to share the power consuming devices among the feeders minimize or eliminate such energy consumption spikes completely.

Let us consider a use case in which an oil terminal has two feeders and multiple power consuming devices (consumers). The number of consumers can range from some

to many. In our use case we have 32 consumers, but in other cases it can be more. These consumers are both pumps and pipeline heating systems. The pumps have a high surge power consumption when starting up which must be taken into consideration when designing an EMS. The EMS monitors the current consumption by polling the consumers via a communication system (e.g., PROFIBUS, CAN bus or Industrial Ethernet). The PROFIBUS communication system is standardized in EN 50170 international standard.

Because the oil terminal stores oil it is considered an explosion hazard area and therefore, a special communication system that is certified for explosive areas - PROFIBUS PA (Process Automation) is used. PROFIBUS PA meets the 'Intrinsically Safe' (IS) and bus-powered requirements defined by IEC 61158-2. The maximum transfer rate of PROFIBUS PA is 31.25 kbit/s which can limit the system response speed if there are many devices connected to the PROFIBUS bus and each device has significant input and output data load.

The EMS is able to switch devices from being supplied from either feeder. Ideally, the power consumption is shared equally among both feeders at all times. This means that the EMS monitors the devices and switches devices over to other feeders if the power consumption is unbalanced among the feeders. In normal operation, the feeder loads are kept sufficiently low to accommodate new devices starting up in a way that the surge consumption will not exceed the threshold power of the feeders.

The EMS polls every power consumer periodically and updates the total consumption. Based on this total consumption, the EMS will command the power distribution devices to switch around from first feeder to the second in case the load on the first feeder is higher than on the second and vice versa.

In our use case we simulate the power consumption of the devices as the input to the IUT. The tester monitors the output (the EMS feeder load values). The test purpose is to verify that neither of the power loads exceed the specified threshold. Exceeding this limit might cause equipment damage and the power company can impose fines upon violating this limit.

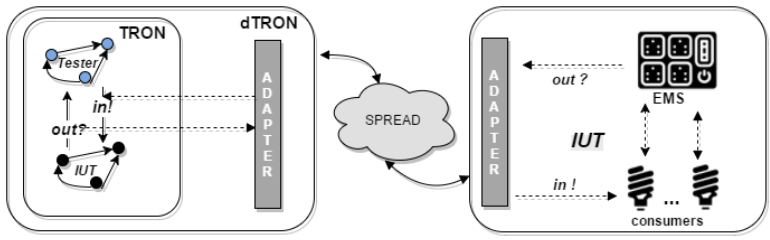


Figure 10. Case Study Test Architecture

The test architecture is depicted in Figure 10. In the right side of the figure, we can see the EMS and consumers as the implementation under test. The test model and test runner is on the left side. The test is executed via DTRON, which transmits the inputs and outputs via Spread. In the IUT and tester models we are going to introduce, the signals prefixed with $i\_$ or $o\_$ are synchronizing signals sent through Spread message serialization service . The signals without the aforementioned prefixes are internal signals which are not published to the Spread network. The input to the IUT is provided by the remote tester model is depicted in Figure 13 which simulates the device power consumption levels and creates challenging scenarios for the EMS. The EMS queries the consumers which are

modeled in Figure 12 and balances the load between the feeders based on the total power consumption monitoring data . The EMS model is shown in Figure 11 which displays the querying loop. The querying is performed in a loop due to the semi-duplex nature of communication in PROFIBUS networks. The EMS also takes the maximum power limit into account as the total power consumption must not exceed this level. This can be seen in the remote tester model shown in Figure 13. Remote tester nondeterministically selects a consumer and sends the level of energy consumption for that particular device to the input port of the IUT. Then the remote tester waits *s* time units before requesting the current feeder energy levels. On the model, it is indicated as *i_get_line_balance*!. After receiving the current values the tester will check whether they are within allowed range. If the values exceed the limit the test verdict is fail. Otherwise the tester will continue with the next iteration.

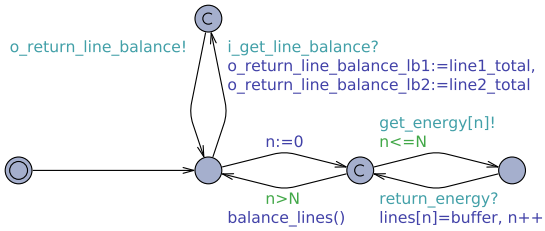


Figure 11. Energy Management System model

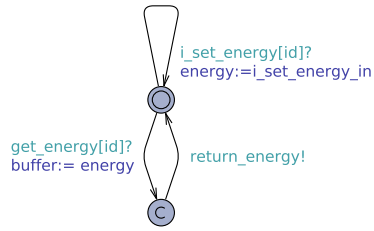


Figure 12. Consumer model

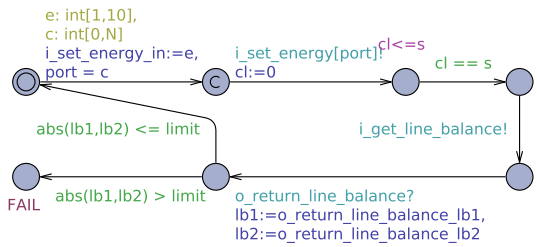


Figure 13. Remote Tester model

The communication delay between receiving the signal from EMS with the current feeder energy levels and sending input to the IUT is 2Δ. According to the specification the system must stabilize the load between feeders in stabilization time limit *s* after receiving the input. If Δ is very close to system stabilization time limit *s* indicated in the remote tester model in Figure 13 the remote tester fails to send the signal in time to the IUT.

For this reason, we introduce the distributed tester Figure 14 where each local component of the tester is closely coupled to the IUT input ports. As shown in chapter 5 this approach reduces the delay by up to Δ. This guarantees that after receiving the output from EMS we can send new input to IUT within less than *s* time units.

7.2. Test Execution Environment DTRON

Uppaal TRON [11] is a testing tool, based on Uppaal engine, suited for black-box conformance testing of timed systems [11]. DTRON [13] extends this enabling distributed

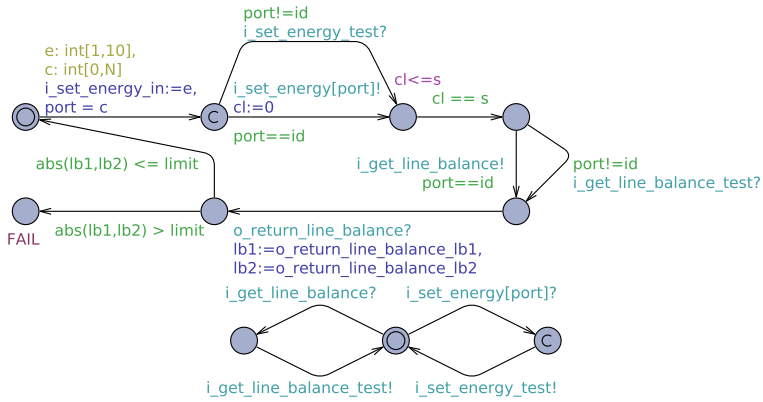


Figure 14. Parametrized local tester template for distributed testing

execution. It incorporates Network Time Protocol (NTP) based real-time clock corrections to give a global timestamp (t_1) to events at IUT adapter(s). These events are then globally serialized and published for other subscribers with a Spread toolkit [18]. Subscribers can be other IUT adapters, as well as DTRON instances. NTP based global time aware subscribers also timestamp the event received message (t_2) to compute and possibly compensate for the overhead time it takes for messaging overhead $\Delta = t_2 - t_1$.

Δ is essential in real-timed executions to compensate for messaging delays that may lead to false-negative non-conformance results for the test-runs. Messaging overhead caused by elongated event timings may also result in messages published in on order, but revived by subscribers in another. Δ can then also be used to re-order the messages in a given buffered time-window t_Δ. Due to the online monitoring capability DTRON supports the functionality of evaluating upper and lower bounds of message propagation delays by allowing the inspection of message timings. While having such a realistic network latency monitoring capability in DTRON our test correctness verification workflow takes into account theses delays. For verification of the deployed test configuration we make corresponding time parameter adjustments in the IUT model.

8. Conclusion

We extend the Δ-testing method proposed originally for single remote tester by introducing multiple local testers on fully distributed test architecture where testers are attached directly to the ports of IUT. Thus, instead of bidirectional communication between a remote tester and IUT only unidirectional synchronization between the local testers is needed in given solution. A constructive algorithm is proposed to generate local testers in two steps: at first, a monolithic remote tester is generated by applying the reactive planning online-tester synthesis method of [16], and second, a set of synchronizing local testers is derived by partitioning the monolithic tester into a set of location specific tester instances. The partitioning preserves the correctness of testers so that if the monolithic remote tester meets 2Δ requirement then the distributed testers meet (one) Δ-controllability requirement. Second contribution of the paper is that distributed testers are generated as Uppal Timed Automata. According to our best knowledge the real time distributed testers have not been constructed automatically in this formalism yet. As for

method implementation, the local testers are executed and communicating via distributed test execution environment DTRON [13]. We demonstrate that the distributed deployment architecture supported by DTRON and its message serialization service allows reducing the total test reaction time by almost a factor of two. The validation of proposed approach is demonstrated on an Energy Management System case study.

References

[1] A. Brook, Evolution and Practice: Low-latency Distributed Applications in Finance. Queue - Distributed Computing, ACM, New York (2015), vol. 13, no. 4, pp. 40-53.
[2] G. Hackenberg, M. Irlbeck, V. Koutsoumpas, and D. Bytschkow, Applying formal software engineering techniques to smart grids. In Software Engineering for the Smart Grid (SE4SG), 2012 International Workshop, IEEE (2012), pp. 50-56.
[3] M. Utting, A. Pretschner, and B. Legeard, A taxonomy of Model-based Testing. Software Testing, Verification & Reliability, John Wiley and Sons Ltd., Chichester, UK (2012), vol. 22, iss. 5, pp. 297-312.
[4] J. Zander, I. Schieferdecker, P. J. Mosterman (eds), Model-Based Testing for Embedded Systems. CRC Press (2011).
[5] ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework - Parts 1-5. International Standard IS-9646. ISO, Geneve (1991).
[6] G. Luo, R. Dssouli, G. v. Bochmann, P. Venkataram, A. Ghedamsi, Test generation with respect to distributed interfaces. Computer Standards & Interfaces, Elsevier (1994), vol. 16, iss. 2, pp.119-132.
[7] B. Sarikaya, G. v. Bochmann, Synchronization and specification issues in protocol testing. In: IEEE Trans. Commun., IEEE Press, New York (1984), pp. 389-395.
[8] R. M. Hierons, M. G. Merayo, M. Núñez, Implementation relations and test generation for systems with distributed interfaces. Springer-Verlag (2012), Distributed Computing, vol. 25, no. 1, pp. 35-62.
[9] A. David, K. G. Larsen, M. Mikuionis, O. L. Nguena Timo, A. Rollet, Remote Testing of Timed Specifications. In: Proceedings of the 25th IFIP International Conference on Testing Software and Systems (ICTSS 2013), Springer, Heidelberg (2013), pp. 65-81.
[10] J. Bengtsson, W. Yi, Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets. LNCS, Springer, Heidelberg (2004), vol. 3098, pp. 87–124.
[11] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, Testing Real-Time Systems Using UPPAAL. In: Hierons, R. M., Bowen, J. P., Harman, M. (eds.) Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. LNCS, Springer, Heidelberg (2008), vol. 4949, pp. 77-117.
[12] G. Behrmann, A. David, K. G. Larsen, A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, Springer, Heidelberg (2004), vol. 3185, pp. 200-236.
[13] DTRON - Extension of TRON for distributed testing, http://www.cs.ttu.ee/dtron.
[14] J. Tretmans: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools, Springer-Verlag (1996), vol. 17, no. 3, pp. 103-120.
[15] R. Segala, Quiescence, fairness, testing, and the notion of implementation. In: Best, E. (eds.) 4th Intrenational Conference on Concurrency Theory (CONCUR'93). LNCS, Springer, Heidelberg (1993), vol. 715, pp. 324-338.
[16] J. Vain, M. Kääramees, M. Markvardt: Online testing of nondeterministic systems with reactive planning tester. In: Petre, L., Sere, K., Troubitsyna, E. (eds.) Dependability and Computer Engineering: Concepts for Software-Intensive Systems, IGI Global, Hershey (2012), pp. 113-150.
[17] A. Anier, J. Vain, Model based Continual Planning and Control for Assistive Robots. In: Proceedings of the International Conference on Health Informatics, SciTePress, Setúbal (2012), pp. 382-385.
[18] The Spread Toolkit, http://spread.org/.

Appendix 2

II

Pal, Deepak; Vain, Jüri (2019). Model based test framework for communications-critical internet of things systems. Databases and Information Systems X: Selected Papers from the Thirteenth International Baltic Conference, DB&IS 2018. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Amsterdam: IOS Press, 79-94

Databases and Information Systems X
A. Lupeikiene et al. (Eds.)

doi:10.3233/978-1-61499-941-6-79

Model Based Test Framework for Communications-Critical Internet of Things Systems

Deepak PAL [1] and Jüri VAIN
Department of Software Science,
Tallinn University of Technology, Estonia

Abstract. In Industry 4.0 large-scale discrete event systems (DES) are becoming increasingly dependent on internet of things (IoT) based real-time distributed supervisory control systems. In order to meet the IoT growing demand, new technologies and skills are being developed which require an automated systematic testing to achieve success in adoption. Testing such systems requires an integration of computation, communication and control in the test architecture. This may pose number of issues that may not be suitably addressed by traditional centralized test architectures. In this paper, a distributed test framework for testing distributed real-time systems is presented, where online monitors (executable code as annotations) are integrated to systems to record relevant events. The proposed test architecture is more scalable than centralized architectures in the sense of timing constraints and geographical distribution. By assuming the existence of a coverage correct centralized remote tester, we give a partitioning algorithm of it to produce distributed local testers which enables to meet more flexible performance constraints while preserving the remote tester's functionality. The proposed approach not only preserves the correctness of the centralized tester but also allows to meet stronger timing constraints for solving test controllability and observability issues. The effectiveness of the proposed architecture is demonstrated by an illustrative example.

Keywords. model-based testing, discrete-event systems, internet of things

1. Introduction

The concept of smart industries came in with the possibility to monitor and control the highly automated production units performance with SCADA (Supervisory Control & Data Acquisition Systems) and DCS (Distributed Control Systems) as widely used smart industry standards. Recently, the Industry 4.0 standard has extended beyond SCADA and DCS to next generation smart factory employing IoT. Emerging industrial IoT applications–smart cities, automotive

[1]Corresponding author, Department of Software Science, Tallinn University of Technology, Akadeemia tee 15A, 19086 Tallinn, Estonia; E-mail: deepak.pal@ttu.ee

and manufacturing etc. require real-time stream analytics and complex event processing to offer optimal–machine utilization, production/maintenance schedules and cost reduction [1]. Today manufacturing setups operate with real-time field data and have grown to the scale of global geographical distribution with numerous services and applications forming an ubiquitous computing network and latency requirements, measured in milliseconds. Further, there are challenges of heterogeneity and environment, where millions of sensors, actuators, and different devices in conjunction with intelligent software forms a complex system and introduce a new dimension to DES real-time testing with significant challenges.

The continuous growth of systems complexity and high demand of security and reliability in the DES has made their testing a big challenge. Moreover, majority of testing and verification techniques have been developed for the non-real-time systems and they cannot be applied on real-time systems due to timing constrains and concurrency issues. Testing DES may pose a number of challenging issues that can not be suitably addressed by traditional centralized remote testing [2]. Major challenges emerge due to severe timing constraints, the tests have to satisfy when the required reaction time of the tester ranges near the message propagation time. These problems restrict the usability of centralized remote testing which has limited capability of controlling of distributed events, and respecting the timing constraints.

For testing DES, designers and developers have frequently used formal verifications techniques during design and development phase of systems [3,4]. In practice, rigorous mathematical proof at the code level is only suitable for small systems due to the state space growth that is exponential in the number of parallel components. Regardless the usage of several state space reduction techniques such as partial order reduction [5] and symbolic model checking [6,7] the problem of scalability still prevents testing and verification of large-scale DES. To address this challenge the idea of online monitoring was proposed in [8,9]. In a distributed system the information communicated to different geographical locations (ports) and their time stamps are not globally known. The lack of holistic view makes the coordination of distributed test agents nontrivial. For online monitoring of the distributed systems several authors [9,10,11] suggested to modify system under test (SUT) to record relevant events (timing and the order of input/output events at different ports) and log the time stamps for global monitoring. The monitored data is collected and integrated to obtain a coherent view of the system. DES augmented with online monitors is a prerequisite of distributed model-based testing (MBT) technique presented in this paper. Online monitoring can be performed inline, in which case the monitors are injected into executable code as annotations [9]. These monitors can be called by applying input to SUT from the location where annotations were placed. However, generating and deploying the monitors for a complex distributed application is a significant engineering effort. Modifying existing distributed systems by instrumenting the annotations may introduce delays and network overhead (probe effect) but there has been lot of research on the implementation of monitors with the aim of obtaining a coherent view of the system and achieving this in a non-invasive manner [9,10,11].

The need for automated online test generation and their correctness assurance have given rise to the use of MBT and the development of several commercial

and academic MBT tools. For instance, smart connected factories with IoT based control systems is a new technology in manufacturing industries which undergoes frequent change in requirement specifications and tools, expecting reduction in testing efforts and costs. In this context, MBT offers an automated tool support (regression testing) and platform independence thus aiming to lower the testing effort of IoT [12]. We interpret MBT in the standard way, i.e. as conformance testing that compares the expected behaviors described by the system model with the observed behaviors of an actual implementation.

In this paper, we propose the online monitors based method for testing DES, where distributed local testers coordinate test activities via SUT which does not require any external network protocol as proposed in [13]. The main assumption is that monitors are injected in non-invasive manner (without interfering the SUT by introducing timing delay, computation/communication overhead, non-determinism, etc). The author of [14] has proposed the testing with monitoring systems using status messages and showed such messages can be used to overcome observability and controllability problems for non real-time systems. We give a partitioning algorithm to produce distributed local testers for real-time systems generated by partitioning the given remote tester model. We assume that there already exists a remote tester generated by applying the reactive planning online-tester synthesis method of [15], and its generation is out of scope of this paper. The proposed approach not only preserves the correctness of the test runs defined in the remote tester but also satisfies the timing constraints for solving controllability and observability issues which might be violated in the centralized testing solution. Further, we sketch first experimental results about our implementations, and describe how it can be used to test smart manufacturing plant with IoT systems.

2. Model Based Distributed Testing

We consider a DES, where a system has to respect timing constraints posed on input/output events. These challenges restrict the capability of centralized remote testing which cannot guarantee the controllability of distributed events, and respect their timing constraints because of message propagation time between the tester and SUT. Another aspect to be considered in DES is that reaching sufficient test coverage by integration testing of such systems in the presence of numerous latency factors and their interdependency, is out of the reach of off-line testing. Since, off-line testing of such systems is not possible due to the non-deterministic nature of SUT off-line testing approaches need to be replaced by on-line distributed testing.

The need for automated online test generation and tests correctness assurance have given rise to the use of MBT. We interpret MBT in the standard way, i.e. as input/output conformance (IOCO) testing that compares the expected behaviors described by the system model with the observed behaviors of an actual implementation. Due to inherent non-determinism of distributed systems the natural choice is online MBT where the test model is executed in lock step with the SUT. The communication between the model and the SUT involves controllable inputs

of the SUT and observable outputs of the SUT which are required for detecting IOCO violations. For detailed overview of MBT and related tools we refer to [16].

2.1. Modelling Implementations of Real-Time Systems

To define our testing architecture formally we need to introduce a semantic foundation for modelling the real-time systems. We describe the notions of timed input output automata (TIOA) introduced [18,17] as a formalism to model the behavior of real-time systems over time. We consider as time domain $\mathbb{T}$ the set $\mathbb{R}_{\geq 0}$ of non-negative reals called clocks (delays) and Σ as a finite set of actions. For the formal syntax and semantics of TIOA we refer the reader to [18].

Example 1: Consider the TIOA specification $\mathcal{S}pec$ shown in Figure 1 (a), $in[1]!$, $in[2]!$, $in[3]!$ denotes the input to the system and $out[1]?$, $out[2]?$, $out[3]?$ denotes the output produced in response to input to the system. The timed $\mathcal{S}pec$ can be expressed in similar language as follow: exactly at 5 time units after the system received the input $in[1]!$ and produces either output $out[3]?$ exactly at 2 time units or, failing to do that , output $out[1]?$ exactly at 3 time units. The clock cl is set to 0 just after passing the transition. A timed trace ρ is a sequence of timed-stamp actions followed with a delay, $\rho_{Seq(\mathcal{R})} = (5 \cdot in[1]!) \cdot (8 \cdot out[1]?) \cdot (15 \cdot in[2]!) \cdot (18 \cdot out[2]?) \cdot 0$. We have $\mathcal{S}pec$ After $(5 \cdot in[1]!) \cdot 0 = \{(l_1, 0)\}$, $\mathcal{S}pec$ After $(5 \cdot in[1]!) \cdot (8 \cdot out[1]?).0 = \{(l_5, 0)\}$ $Out(\mathcal{S}pec$ After $(5 \cdot in[1]!) \cdot (7 \cdot out[1]?).0) = \mathbb{T}$, $Out($ $\mathcal{S}pec$ After $(5 \cdot in[1]!) \cdot (8 \cdot out[1]?).15) = \{in[2]!\} \cup \mathbb{T}$.

Uppaal Timed Automata (UTA): In our approach UTA [19,18] are used as a formalism to illustrate TIOA to model SUT behavior. This choice is motivated by the need to test the SUT with timing constraints so that the impact of propagation delays between the SUT and the tester can be taken explicitly into account when the test cases are generated and executed. UPPAAL is based on the definition of timed automata, which is introduced by [17]. For the full formal syntax and semantics of UTA we refer to [19,18].

Modelling distributed n-Ports: We model a multi-ports TIOA in UTA by splitting the transition with multiple communication actions to a sequence of transitions each labeled with exactly one I/O-action and connected via committed locations, so that all ports of such group are updated instantaneously in the order they are specified in the tuple. In Figure 2 the labels on the transition represent the i/o actions and the transition tuple (l_0, l', $in[1]!$ /($out[1]?$, $out[3]?$))

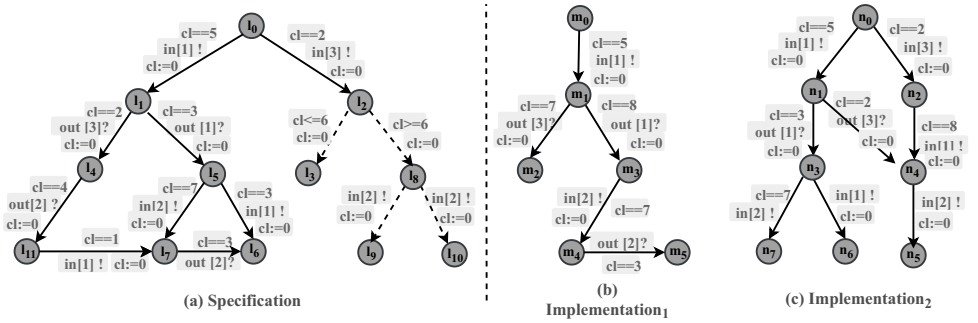


Figure 1. Models of TIOA specification and implementations

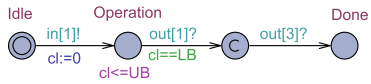


Figure 2. Modelling pattern of multiport timed automata

is represented by sequence of transitions each labeled with exactly one action and connected via committed locations, l_0 represents the *idle*, and l' represents the *Done* location. Let P_{l_n} denotes a set of ports accessible in the physical location l_n where $n \in \mathbb{N}$; I is a n-tuple (I_1, I_2,...,I_n), where I_i is the finite set of inputs at port i, $I_i \cap I_j = \phi$ for $i \neq j$ and $i,j = 1,...n \in \mathbb{N}$. Similarly, O is a n-tuple (O_1, O_2,...O_n), where O_i is finite set of outputs at port i, $O_i \cap O_j = \phi$ for $i \neq j$ and $i,j = 1,...n \in \mathbb{N}$. Each port may receive outputs of other port, i.e $O = (O_1 \cup \{\varepsilon\}) \times (O_2 \cup \{\varepsilon\}) \times ... \times (O_n \cup \{\varepsilon\})$, here $\{\varepsilon\}$ denotes the empty output in response to input to $\mathcal{SUT}$.

2.2. Timed Input/Output Conformance Relation

In order to define the conformance relation, we recall the timed input/output conformance relation (`tioco`) introduced in [20,21]. They propose extension of `ioco` relation with timing constraints including clock valuations with the set of observable actions. The communication between the specification and the SUT involves controllable inputs of the SUT and observable outputs of the SUT. In this work, we introduce the `ioco` at first as defined by [22]. The behaviour of `ioco`-correct implementation should respect after some observations following restrictions: *(i)* the outputs produced by SUT should be the same as allowed in the requirements model;*(ii)* if a quiescent state (a situation where the system cannot evolve without an input from the environment) is reached in SUT, this should also be the case in the model; *(iii)* any time an input is possible in the model, this should also be the case in the SUT. In addition to `ioco`, `tioco` introduces the time delays observable on test interface. This is explained by means of following example (for detailed definition of (`tioco`) we refer to [20,21]. *Example 2:* Consider the timed I/O automata specification $\mathcal{S}pec$ and implementations $\mathcal{I}mpl_1$, $\mathcal{I}mpl_2$ shown in Figure 1. Based on `tioco` relation, we can verify that if $\mathcal{I}mpl_1$ conforms to $\mathcal{S}pec$, for example: $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]!)) = \mathbb{T}$ and $Out(\mathcal{I}mpl_1 \text{ After } (5 \cdot in[1]!)) = \mathbb{T}$; $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]!) \cdot 8) = \{out[1]?\} \cup \mathbb{T}$ and $Out(\mathcal{I}mpl_1 \text{ After } (5 \cdot in[1]!) \cdot 8) = \{out[1]?\} \cup \mathbb{T}$; $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]!) \cdot 7) = \{out[3]?\} \cup \mathbb{T}$ and $Out(\mathcal{I}mpl_1 \text{ After } (5 \cdot in[1]!) \cdot 7) = \{out[3]?\} \cup \mathbb{T}$, proves that $\mathcal{I}mpl_1$ `tioco` $\mathcal{S}pec$. Similarly, we can prove that $\mathcal{I}mpl_2$ ~~`tioco`~~ $\mathcal{S}pec$ i.e. $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]!) \cdot (7 \cdot out[3]?)) = \mathbb{T}$ and $Out(\mathcal{I}mpl_2 \text{ After } (5 \cdot in[1]!) \cdot (7 \cdot out[3]?)) = \{in[2]!\} \cup \mathbb{T}$; $Out(\ \mathcal{S}pec \text{ After } (5 \cdot in[1]!) \cdot (8 \cdot out[1]?).18) = \{out[2]?\} \cup \mathbb{T}$ and $Out(\mathcal{I}mpl_2 \text{ After } (5 \cdot in[1]!) \cdot (8 \cdot out[1]?).18) = -$.

3. Challenges of Centralized Remote Testing

In [2], authors addressed the conformance testing of remote SUTs and proposed the testing architecture which is composed of one FIFO for each direction of com-

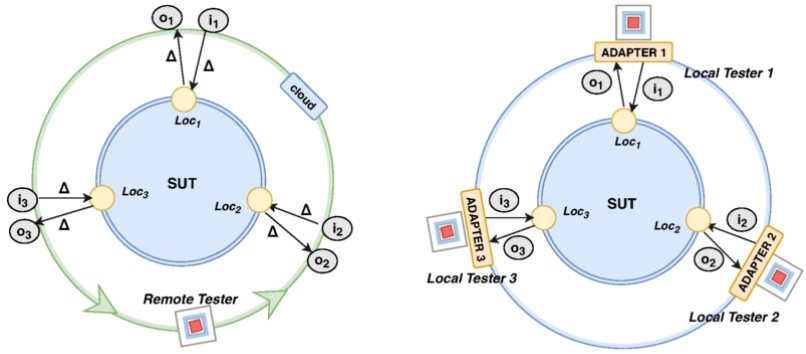
Table 1. A time trace observed by SUT and Remote Tester

$\rho_{\mathcal{SUT}}$:	$(5 \cdot in[1]!) \cdot (7 \cdot out[3]?) \cdot (11 \cdot out[2]?) \cdot (12 \cdot in[1]!) \cdot (15 \cdot out[2]?)$
$\rho_{\mathcal{S}pec}$:	$(2 \cdot in[1]!) \cdot (9 \cdot in[1]!) \cdot (10 \cdot out[3]?) \cdot (14 \cdot out[2]?) \cdot (18 \cdot out[2]?)$

munication between SUT and remote tester with communication latency bounded by Δ. Using Δ-testability criteria, it was shown that if the SUT ports are remotely observable and controllable then 2Δ-condition is sufficient for satisfying timing correctness of the test. Here, Δ denotes an upper bound of message propagation delay between tester and SUT ports. Though this approach works reasonably well for systems with sufficient timing margins, it cannot be extended to systems with the timing constraint less than 2 Δ. This means that the actions may not reach the port in time and as a result, the testing becomes infeasible in such systems.

Impact of latency in remote testing: In remote testing the reactions are not always received in the order their stimuli are sent. In order to control the simultaneous test inputs, tester should not wait to receive outputs before sending the next input to $\mathcal{SUT}$. *Example 3*: Consider the timed I/O automata specification $\mathcal{S}pec$ in Figure 1 (a). Assume that the propagation latency exactly 3 time units between $\mathcal{SUT}$ and $\mathcal{S}pec$, which means if $\mathcal{S}pec$ has to apply input to $\mathcal{SUT}$, it should send that input 3 time units earlier, so that $\mathcal{SUT}$ receive the input on time as specified in specification. To maintain the propagation delay, the $\mathcal{SUT}$ and tester shall observe the timed trace shown in Table 1. The $\mathcal{S}pec$ sends second input $in[1]!$ at 9 time units before receiving outputs $out[3]?$, and $out[2]?$ in response to previous input $in[1]!$ at 2 time units to $\mathcal{SUT}$. It seems, outputs $out[3]?$, and $out[2]?$ are generated in response to second input $in[1]!$, though $\mathcal{SUT}$ produces outputs as specified in $\mathcal{S}pec$ and sends $out[3]?$, and $out[2]?$ to $\mathcal{S}pec$ before receiving the second input $in[1]!$. However, the emission of an second input $in[1]!$ depends on the reception of an outputs $out[3]?$, and $out[2]?$, because of latency and to maintain it, tester should not wait to receive outputs before sending the input to $\mathcal{SUT}$. This means in remote testing the propagation latency between $\mathcal{SUT}$ and $\mathcal{S}pec$ may lead to unintended interleaving of input/output actions. This affects the generation of inputs for the $\mathcal{SUT}$ and the observation of outputs that may trigger a wrong test verdict.

Consider the remote testing architecture depicted in Figure 3(a) and its corresponding UTA model in Figure 4. The SUT shown in figure has 3 ports (p_1, p_2, p_3) in geographically different places with inputs/outputs $in[1]/out[1]$, $in[2]/out[2]$ and $in[3]/out[3]$ at ports $p1$, p_2 and p_3 respectively. The UTA models defines the expected global behavior of any potential SUT. Each expected global behavior is expressed as the sequence of labels of UTA model edges. This global trace is transformed to the *global test sequence*. Another important aspect that needs to be addressed in remote testing is functional non-determinism of the SUT behaviour with respect to test inputs. For non-deterministic systems only online testing (generating test stimuli on-the fly) is applicable in contrast to that of deterministic systems where test sequences can be generated offline. The source of timing nondeterminism in remote testing of real-time systems is communication latency between the tester and the SUT that may lead to interleaving of inputs and outputs discussed in *Example 3*. Consequently, the centralized remote testing approach is not suitable for testing a real-time distributed system if the system



(a) Centralized (b) Distributed

Figure 3. Centralized vs Distributed test architecture

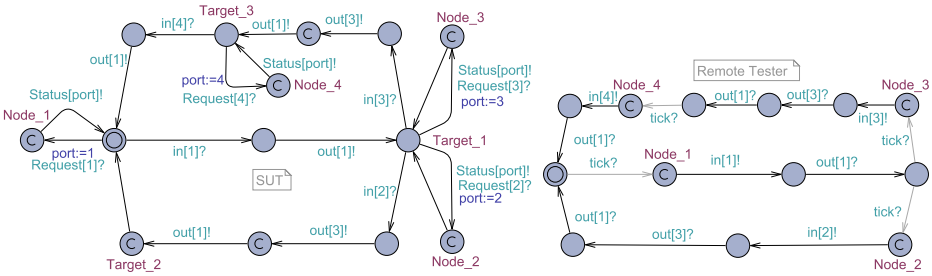


Figure 4. SUT and Remote tester models

has strict timing constraints that require reactions faster than 2 Δ. The shortcomings of the centralized remote testing approach are mitigated with overcoming the 2 Δ constraint by partitioning the remote tester into multiple local testers as shown in Figure 3(b).

4. Distributed Testing

4.1. Distributed Test Architecture

An alternative to the remote testing is distributed testing the architecture of which is shown in Figure 3(b). Here the remote tester model deployed on a centralized testing architecture is decomposed into a set of communicating (via SUT) local testers, one for each localized interface of the system. Those localized testers communicate with the system by means of adapters whose purpose is both to transfer data between the local interfaces and the localized testers. The approach is supposed to reduce the latency in communication between SUT and local testers (*SUT receiving input and local testers receiving outputs*) caused by two-way messages between the remote tester and SUT. Since the local testers are attached at the same sites as the test ports of SUT the communication delay between a local tester adapter and the SUT port is negligible instead of bidirectional communication needed in case of remote tester.

To implement the model level synchronization of local testers, e.g. to force two test inputs to be inserted at different ports of remote locations at the same time (in the sense of model time), our distributed test execution environment DTRON [23] implements these synchronization channels between local tester models using Spread services [24]. Deterministically controllable test run presumes output observability, which means that the tester attached to some port is waiting for the expected output from this port and after receiving it, propagates it to other testers whose behaviour depends on it. In conventional remote tester case the test stimulus travels from the tester to some SUT port in one Δ and the response from the SUT back to the tester takes another Δ (bidirectional communication), while in the distributed tester's communication with local ports one Δ can be saved.

4.2. Centralized Tester Partitioning Algorithm

We apply *Algorithm 1* to transform the centralized remote tester depicted in Figure 3(a) into a set of communicating distributed local testers, the architecture of which is shown in Figure 3(b). In this paper we considered a simple case of remote tester model to provide a better theoretical understanding of algorithm. This algorithm covered few cases where tester expects at least one possible output in response to applied input. Line 2-11 adds an adapter model to each local tester instance. The purpose of adding an additional adapter instances to each local tester instance is that it synchronize the local communication between $\mathcal{SUT}$ local ports and a local tester. Its model is derived from remote tester model by adding original channels of $\mathcal{SUT}$ and by renaming channels of local testers. For clarity, notations T_l and A_l represents local tester and local adapter respectively. The channel `tick` denotes the one clock tick (where timing constraints are encoded in `clock Tick Gen` model), each tick represents the real clock variable which track the time elapsed. It provides the time frame for apply input and wait for receiving output from SUT in the same tick.

We assumed that monitors are injected at each port of SUT which can be called by applying special input to SUT from the port where annotations were placed. The Channel `Request[`l_n`]` represents the special input to SUT that leads to an output `status[`*port*`]` being sent to all SUT-ports and used to coordinate the other local testers via SUT, here argument *port* represents the location from where the `Request[`l_n`]` is being applied. An output `status[`*port*`]` generated in response to input `Request[`l_n`]` provides the coherent view of the system to simplify distributed testing and helps local testers to synchronize with each other. The channel `status[`*port*`][`l_n`]` represents the local communication between adapter and tester. The input `in_[`l_n`]` and output `out_[`l_n`]` are the channels between local adapter and local tester. The ***chan*** `in_[`l_n`]?` represents the reception `R` of input i.e. $T_l \xrightarrow{\texttt{in\_[l}_\texttt{n}\texttt{]?}} A_l$ and ***chan*** `out_[`l_n`]!` represents the emission `E` of output i.e. $A_l \xrightarrow{\texttt{out\_[l}_\texttt{n}\texttt{]!}} T_l$. Similarly, the channels `in[`l_n`]? out[`l_n`]!` are the channels between $\mathcal{SUT}$ and adapter. Now, the construction of local testers, for each locations l_n, we take clone of M^{RT} to be transformed into a location specific local tester instance M^{l_n} (Line 12). The loop in Line 13 says for each clone testers model M^{l_n}, we go through all the edges i/o pair. For clarity, we divided the distribution into two cases, in Line 16, *Case 1* says if the edge has a synchronizing channel i.e

Algorithm 1 Automated Construction of Adapter and Local Testers

1: *input*: M^{RT}; *output*: $||_n M^{\mathcal{DT}}$ $n \in \mathbb{N}$;
2: **for all** $l_n \in Loc(\mathcal{SUT})$ **do** ▷ $n \in \mathbb{N}$
3: *Add* ***chan*** `tick?` ▷ R: $Clock\_Gen \rightarrow A_l$
4: *Add* ***chan*** `Request[`l_n`]!` ▷ E: $A_l \rightarrow \mathcal{SUT}$
5: *Add* ***chan*** `in_[`l_n`]?` ▷ R: $T_l \rightarrow A_l$
6: *Add* ***chan*** `in[`l_n`]!` ▷ E: $A_l \rightarrow \mathcal{SUT}$
7: *Add* ***chan*** `status[port]?` ▷ R: $\mathcal{SUT} \rightarrow A_l$
8: *Add* ***chan*** `status_[port][`l_n`]!` ▷ E: $A_l \rightarrow T_l$
9: *Add* ***chan*** `out[`l_n`]?` ▷ R: $\mathcal{SUT} \rightarrow A_l$
10: *Add* ***chan*** `out_[`l_n`]!` ▷ E: $A_l \rightarrow T_l$
11: **end for**
12: *copy* M^{RT} to M^{l_n} ▷ take clone at each location
13: **for all** M^{l_n}, $n \in \mathbb{N}$ **do**
14: **for all** `chan[`l`]: in[`l_n`]/out[`l_n`]` pairs $\in M^{l_n}$ **do**
15: *Case 1*: Consider arbitrary port, n = i $\in l_n$
16: **if** edge.`in[`i`]` $\wedge$ edge.`out[`i`]` $\wedge$ $i \in P_{l_i}$ **then**
17: *Add* ***chan*** `status_[port][`i`]?` ▷ R: $A_l \rightarrow T_l$
18: *Rename* `chan in[`i`]!, out[`i`]?` to `in_[`i`]!, out_[`i`]?`
19: **end if**
20: *Case 2*: Consider arbitrary port, n = j $\in l_n$
21: **if** edge.`in[`j`]` $\wedge$ edge.`out[`j`]` $\wedge$ $j \notin P_{l_i}$ **then**
22: *Replace* `chan in[`j`]!, out[`j`]?` to `status_[`j`][`i`]?` ▷ R: $A_{l_j} \rightarrow T_{l_i}$
23: **end if**
24: **if** edge.`in[`j`]` $\wedge$ edge.`out[`j`]/out[`i`]` *where* $j \notin P_{l_i} \wedge$ `out[`i`]` $\in P_{l_i}$ **then**
25: *Replace* `chan in[`j`]!, out[`j`]?` to `status_[`j`][`i`]?` $\rightarrow$ edge.`out[`i`]`
26: **end if**
27: **end for**
28: **end for**

M^{RT}: Remote Tester Model; $||_n M^{\mathcal{DT}}$: Communicating Distributed Testers; $l_n \in Loc(\mathcal{SUT})$: represents the number of ports of SUT; *Clock_Gen*: timing constraints encoded in model; *R*, *E*: represents Reception and Emission of channel; A_l: Adapter Model; T_l: Local Tester Model;

`in[`l_n`]/out[`l_n`]` and the channel belongs to same port location $l_n \in P_{l_n}$ then we add the reception (co-action) of chan `status[`*port*`][`l_n`]` and *Rename* the ***chan*** `in[`l_n`]!, out[`l_n`]?` to `in_[`l_n`]!, out_[`l_n`]?` as shown in Figure 5. Basically, idea is to minimize the automata M^{l_n} by removing all synchronizing channels that do not belong to this location. In Line 20, *Case 2* says if input `in[`l_j`]` $\wedge$ and output `out[`l_j`]` does not belong to same port location $l_j \notin P_{l_i}$ then replace those channels with channel `status[`l_j`][`l_i`]`. Similarly, in Line 24, if there is an output `out[`l_i`]` generated by SUT in response to input `in[`l_j`]`, channel `status[`l_j`][`l_i`]` is followed by `out[`l_i`]`. Figure 5 represents the generated parameterized local tester with corresponding parameterized adapter model where `L` denotes the geographical location.

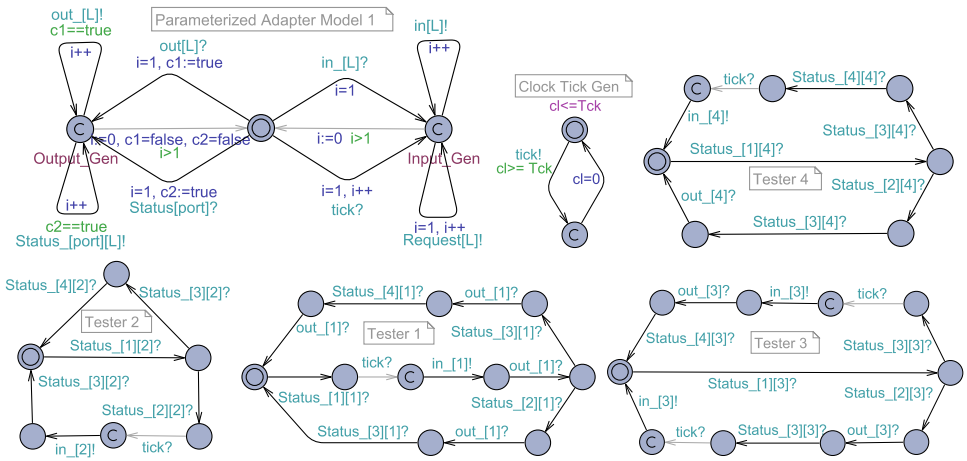


Figure 5. Parameterized adapter and local tester models

In remote testing, tester generates an input for the SUT, waits for the result and continues with the next set of inputs and outputs until the test scenario has been finished. Thus, the tester has to wait for the duration it takes the signal to be transmitted from the tester to the SUT's ports and the responses back from ports to the tester. Therefore, $\mathcal{SUT}$ conforms M^{RT} if following constraints are satisfied: *1. Order constraint* In Figure 4, remote tester can generate inputs $in[2]!$ (`Node_2`) or $in[3]!$ (`Node_3`) only after receiving $out[1]?$ (`Node_1`) in response to applied input $in[1]!$; *2. Timing constraint* In the case of SUT being distributed in a way that signal propagation time is non-negligible, this can lead into a situation where the tester is unable to generate the necessary input for the SUT in time due to message propagation latency. For example, if the inputs $in[2]!$ (`Node_2`) or $in[3]!$ (`Node_3`) has clock constraints and input $in[1]!$ (`Node_1`) executed before them and waiting to receive output from SUT, then the delay separating two consecutive inputs must satisfy the condition clock constraints at tester and propagation latency $\Delta \leq$ max waiting time by SUT.

Let us consider now, $\mathcal{SUT}$ conforms $_{\mathbb{H}n}$ $M^{\mathcal{DT}}$ if order constraints on observable actions and timing constraints are satisfied. To prove both the constraints on actions applied by local testers, we have following test sequence: *(i)* We consider a clock constraint encoded in the tick model, where each action triggered has to finish execution in one tick, where a tick is synchronized with all local tester models. Timings constraints encoded in the model are fictitious and must be respected by real SUT; *(ii)* We assume that tester apply input `Request[`l_i`]` (before actual input) to SUT that leads to an output `status[`i`]` sent to all SUT-ports in one tick and reset the clocks; *(iii)* Immediately after executing the `Request[`l_i`]`, the tester sends $in[l_i]!$ to SUT. This may lead to an output at all port with in one tick cycle. Any output actions observed outside the time frame is consider as non-conformance with SUT. We show that problem of controllability and observability in DES can be overcome by testing systems with timing constraints, provided monitors (input `Request[`l_i`]` and Outputs `status[`l_i`]`) are implemented correctly.

Observability Problem: The input `Request[`l_i`]` applied to SUT results in output `status[`l_i`]` to all ports. Output `status[`l_i`]` has sufficient information and allows the other local testers to guarantee the order and timing constraints of incoming input/output actions from port l_i. Each local tester recognizes the port where input was applied for which the specific output response was received at their ports, that overcomes the observability problems among local testers.

Controllability Problem: Similarly, if $tester_j$ at location j has to apply input after input $in[l_i]!$ triggered by $tester_i$ at location i. The $tester_j$ has to wait for following actions: For $tester_j$ to apply input upon execution of input from $tester_i$, it wait for reception of output `status[`l_i`]` in response to `Request[`l_i`]` and output $out[j]?$ (if any) generated by SUT in response to input $in[i]!$ from location i. As we assumed that communication delay between testers and SUT is negligible, it eliminates the possibility of introducing delay and overlapping messages with others. Also clock constraints encoded in tick model force to respect timing constraints. Hence, waiting for the reception of `status[`l_i`]` and output $out[j]?$ allows the $tester_j$ to overcome problem of controllability.

5. Case Study: MBT for Smart Manufacturing Plant with IoT Systems

We study the control and coordination problem of large-scale distributed DES in the presence of subsystems where the communication delay between subcomponent (actuators, sensors, control) can influence the behavior of IoT systems. In use case, we have SUT that has four ports geographically distributed at different locations as shown in Figure 6. These ports represent sensors/actuators and IoT system interface that are geographically located in different places. In such a situation, the propagation of the input and output signals are not negligible and affects the distributed system process. Each port consists of inputs and outputs, but not necessarily both. The inputs of the port represent data input to the subcomponent and output is the output from the subcomponent. In plant side, these represent the events detected by sensors e.g. job start and stop times and idle time of machine, and in IoT side these represents the data analytics (machine utilization, energy prices, renewal resources, production constraints) generated by IoT which are used by plant supervisors. Each geographical location denotes a heterogeneous environment (adapters, connectors, wireless sensor/actuators appliances) with a variety of machines. Each appliance consists of a computational component and sensor that sense the real-time events (job starts, stop times) performed on the machine and transmitted to the IoT systems via central controller. Besides events from plant, real-time forecast information from other stakeholders such as production constraints (no. of workpiece requirements per day), dynamic electricity pricing (include real-time and critical-peak pricing) constraints and weather forecasting etc, are also received by the IoT systems which aggregate all these information and perform analytics on them.

To prove the advantages of distributed test architecture over conventional centralized tester and show applicability of distributed testers, we consider a worst case scenario to test (load/performance) the IoT systems (capable of handling massive data from different components in the system leads to conflicts/denial-of-

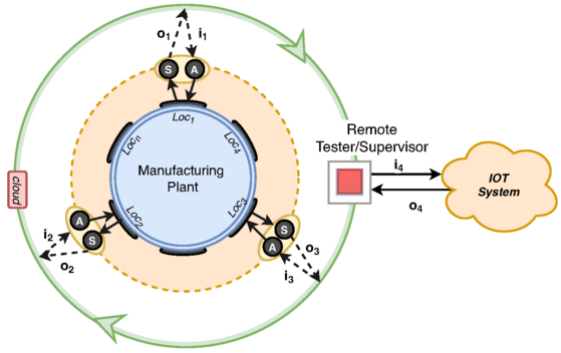


Figure 6. IoT Based DES Architecture for Centralized Controller

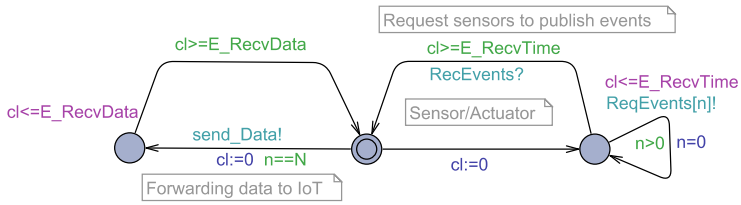


Figure 7. Centralized Tester Communication Model

service attack). Now, testing such scenario is out of scope of conventional remote tester when SUT has strict timing constraints. As we know that the remote tester generate a data input for the SUT (request to sensor/actuator), waits for the result (real-time events), tester has to wait for the duration it takes the signal to be transmitted from the tester to the SUT's ports and the responses back from ports to the tester. It seems impossible to perform performance testing of IoT systems to handle bombarding of events without conflicts at the same time.

Using distributed test architecture augmented with online monitors, we can deploy the communicating local testers generated from algorithm at each port of SUT (representing location of sensors/actuators). Most IOT systems use more than one sensors to perform a tasks, component attached to the sensor may depend on other sub-component of the system. Moreover to finish a tasks, an application usually requires to access data from multiple sensors. A separate standalone netowrk for managing critical communication among sensor can cause extra delays in test architecture, thus, it is impractical to implement such standalone management system for each sensor. For effective and optimized test architecture, we assumed that monitors are injected at each sensor locaion of SUT to handle real-time critical communication among them. Using monitors, each local tester can coordinate each other and can generate the worst case scenario where system can be test against bombarding of real-time events from SUT ports. We start modelling the test scenario, at first, with specifying remote locations and i/o ports of those shown in Figure 6. After applying the algorithm 1, the centralized remote testing architecture depicted in Figure 7 is transformed into distributed testing architecture depicted in Figure 8. In distributed testing archi-

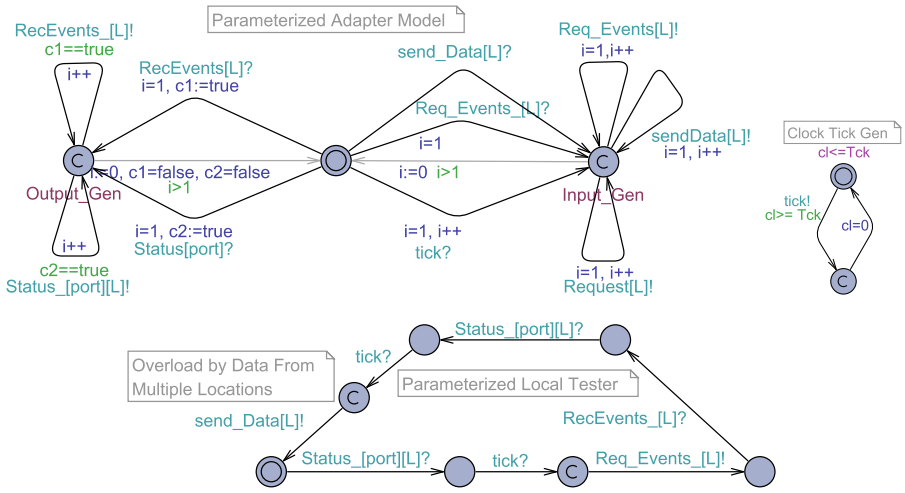


Figure 8. Parameterized Distributed Local Tester and Adapter Models

tecture all distributed location (i/o ports with sensors/actuators) of plant and IoT system all together form the SUT. As can be seen from the Figure 6, the parallel bidirectional communication channels `i/o` between the centralized remote tester and distributed SUT locations (sensor/actuators locations and IoT) have been eliminated as the centralized tester is split into distributed local instances and attached directly to the SUT locations. The corresponding local testers at the these locations with their adapter model are depicted in Figure 8. As discussed in section 5, this results in reduced message propagation timing needs and enables testing a real-time distributed system under the timing constraints close to the message propagation time range.

6. Related Work

In this section, we review work related to our proposed approach in the context of MBT of distributed DES with integrated IoT platform. Substantial work has been done in the control communities, where the centralized and modular methods have been proposed for the synthesis of a supervisory control [25,26,27,28,29]. Also, the challenges of fault diagnosis in DES have been addressed with both model-based (where expected/observed behavior is compared against formal model) [30, 31,32,33,34] and non-model based methods (where observed/expected behavior matched with known faults) [35,36,37]. It is known that modular(distributed) synthesis for DES can render better design flexibility than centralized synthesis. To avoid state space explosion issues in centralized approach, distributed methods (n supervisor controls) have been used. To the best of author's knowledge, the literature lacks testing methods for real-time distributed supervisory control. As the vision of industry 4.0 is to extend beyond SCADA and DCS to next generation smart factory employing IoT, the systems need to be carefully verified to address challenges of time constraints, scalability and concurrency to achieve success in their adoption. Moreover, frequent change in requirement specifications and re-tooling expects rigorous testing of the system.

MBT has been extensively studied in the literature [38,39]. However, the majority of existing methods in IoT domain are specifically designed for testing cloud-based application. It was investigated in [40] that testing for IoT and its application today remains still a challenge. As the IoT technologies are growing in DES, issues of concurrency, scalability, and real-time constraints imposed by the design of real-time systems need to be addressed. As for the state of the art in the following, we review the existing work in testing distributed systems and extend it in the field of MBT related to DES and IoT systems.

As for broader context of distributed testing the early works focused on testing distributed non real-time systems [41,15,42]. The theory of testing distributed real-time systems has gained interest only in recent years when global time keeping techniques emerged. For instance the authors of [43] assumed that each local tester has a local clock which adds timestamps to its observations. It is assumed that the proposed approach provides additional information regarding the causality between actions performed at different ports. But the approach relies on strong assumption that it is known how much local clocks can differ between synchronization events which appears to be unrealistic. Though approach works reasonably well for distributed testing it cannot be used if specifications contains strict timing requirements, especially if there are requirements regarding the relative timing of actions at different ports. Pioneering results on testing timing correctness with remote testers was proposed in [2] where a remote abstract tester was proposed for testing distributed systems in a centralized manner. It was shown that if the SUT ports are remotely observable and controllable then 2Δ-condition is sufficient for satisfying timing correctness of the test. Here, Δ denotes an upper bound of message propagation delay between tester and SUT ports. Though this approach works well for systems with sufficient timing margins, it cannot be extended to systems with the timing constraint more strict than $2\ \Delta$. This means that the test inputs may not reach the input port in time and as a result, the testing becomes infeasible in such systems. The shortcomings of the centralized remote testing approach are mitigated by partitioning the remote tester into multiple local testers that are deployed in the same locations with the SUT component they are testing [13] where the controllability and observability problems are resolved by allowing the local testers to exchange messages through external reliable communication independent of the SUT.

7. Conclusion

In this paper, a distributed test framework for testing of a DES augmented with monitors is presented, where online monitors are used to record relevant events (timing and order of input/output events at test interface ports). Online monitored data is used to obtain a coherent view of the system and to simplify distributed testing, where local testers synchronize with each other via communicating with these monitor. The proposed test architecture is test reaction time wise more scalable than centralized remote test architecture for testing large number of geographical locations (ports) in a system. We give a partitioning algorithm to produce automated distributed local testers from given remote tester model. The

proposed approach not only preserves the functional correctness of the centralized remote testers but also satisfy stronger timing constraints needed for solving distributed test controllability and observability issues.

References

[1] Altizon, Industry 4.0 vision of Smart Factory: Extending beyond SCADA (Supervisory Control & Data Acquisition Systems) and DCS (Distributed Control Systems) | IoT-SCADA Industry 4.0, https://altizon.com/iot-scada-complementary-technologies/

[2] David, A., Larsen, K. G., Mikuionis, M., Nguena Timo, O. L., Rollet, A: Remote Testing of Timed Specifications,In: Proceedings of the 25th IFIP International Conference on Testing Software and Systems, pp. 6581. Springer, Heidelberg (2013).

[3] Wachter, B., Genon, A., Massart, T., Meuter, C.: The formal design of distributed controllers with dSL and Spin, Formal Aspects of Computing", pp. 177–200. (2005)

[4] Cimatti, A., Clarke, M., Enrico, G., Fausto, G., Marco, P., and Armando, T.: Nusmv 2: An Open Source Tool for Symbolic Model Checking, In: CAV, (2002).

[5] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, vol. 1032 Springer, (1996).

[6] Clarke, E., Grumberg, O., and Peled. D.: Model Checking. The MIT Press, (1999).

[7] McMillan, K. L.: Symbolic model checking: an approach to the state explosion problem. Carnegie Mellon University, (1992).

[8] M. Leucker and C. Schallhart: A brief account of runtime verification. Journal of Logic and Algebraic Programming, (2008).

[9] Goodloe, A., Pike, L.: Monitoring distributed real-time systems: a survey and future directions (NASA/CR-2010-216724), Havelund, K., Rosu, G.: Synthesizing monitors for safety properties, (2010).

[10] A. Bauer, M. Leucker and C. Schallhart: Model-based runtime analysis of distributed reactive systems, Australian Software Engineering Conference, pp. 10. (2006).

[11] K. Sen, A. Vardhan, G. Agha and G. Rosu: Efficient decentralized monitoring of safety in distributed systems, In: Proceedings of 26th International Conference on Software Engineering, pp. 418-427. (2004).

[12] Reinhart Richter, Xcerra Corporation, Does the Internet of Things force us to rethink our test strategies? http://xcerra.com/ep doestheinternetofthingsforceustorethinkourtest-strategiesvision

[13] Vain, J., Halling, E., Kanter, G., Anier, A., Pal, D: Model-based testing of real-time distributed systems. 12th International Baltic Conference on Databases and Information Systems, pp. 1-14. Riga, (2016).

[14] R.M. Hierons: Using status messages in the distributed test architecture, Information and Software Technology, Volume 51, Issue 7, pp. 1123-1130. (2009).

[15] Vain, J., Kaaramees, M., Markvardt, M.: Online testing of nondeterministic systems with reactive planning tester, Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 113–150. Hershey (2012).

[16] Utting, M., Pretschner, A., and Legeard, B: A taxonomy of Model-based Testing, Software Testing, Verification & Reliability, J. Wiley and Sons Ltd., UK, (2012).

[17] R. Alur and D. Dill: A theory of timed automata. Theoretical Comp. Science, (1994).

[18] Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools, In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets, vol. 3098, pp. 87–124. (2004).

[19] Behrmann, G., David, A., Larsen, K. G.: A Tutorial on Uppaal, In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, vol. 3185, pp. 200–236. (2004).

[20] Krichen, M., Tripakis, S.: Black-box conformance testing for real-time systems. In: 11th International SPIN Workshop on Model Checking Software. Vol. 2989 of Lecture Notes in Computer Science., pp. 109126. (2004).

[21] Mikucionis, M., Larsen, K.G., Nielsen, B.: T-uppaal: Online model-based testing of real-time systems. In: 19th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, pp. 396397. (2004).
[22] Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools, vol. 17, no. 3, pp. 103–120. (1996).
[23] Anier, A ., Vain, J ., Tsiopoulos, L.: DTRON: A Tool for Distributed model-based testing of time critical applications. Estonian Academy of Sci., pp. 75-88. (2017).
[24] The Spread toolkit. http://spread.org/ (accessed 2016-11-24).
[25] L. Ouedraogo, M. Nourelfath and A. Khoumsi, A new method for centralized and modular supervisory control of real-time discrete event systems, 2006 8th International Workshop on Discrete Event Systems, Ann Arbor, MI, 2006, pp. 168-175.
[26] A. Khoumsi, Coordination of Components in a Distributed Discrete-Event System, The 4th ISPDC05, Lille, 2005, pp. 299-306.
[27] A. Vahidi, M. Fabian, B. Lennartson, Efficient supervisory synthesis of large systems, Control Engineering Practice, vol.14 (10), pp.1157-1167, 2006.
[28] J. Komenda, and J. H. Van Schuppen, Modular Control of Discrete-Event Systems with Co-algebra, IEEE Trans. Autom. Control, vol.53, no.2, pp.447-460, 2008.
[29] K. Cai and W. M. Wonham, New results on supervisor localization, with case studies, Discrete event Dyn. Syst., vol.25, no.1, pp.203-226, 2015.
[30] R. K. Boel and J. H. van Schuppen, Decentralized failure diagnosis for discrete-event systems with costly communication between diagnosers, Sixth International Workshop on Discrete Event Systems, 2002. Proceedings., 2002, pp. 175-181.
[31] G. Provan, Diagnosability analysis for distributed systems, Proceedings, IEEE Aerospace Conference, 2002, pp. 6-2943-6-2951 vol.6.
[32] A. Schumann, Y. Pencole, and S. Thiebaux, Diagnosis of discrete-event systems using binary decision diagrams, in Proceedings of the Fifteenth International Workshop on Principles of Diagnosis (DX04), (2004).
[33] L. Trave-Massuy, T. Escobet, and R. Milne, Model-based diagnos ability and sensor placement application to a frame 6 gas turbine subsystem, in Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, volume 1, pp. 551556, (2001).
[34] Y.Pencole, M.O. Cordier, and L. Roz e, A decentralized model-based diagnostic tool for complex systems, International Journal on Artificial Intelligence Tools, 11(3), 327346, (2002).
[35] R. Sengupta, Diagnosis and communication in distributed systems, in Proceedings of the Workshop on Discrete Event Systems (WODES98), pp. 144151, Cagliari, Italy, (1998).
[36] R. Debouk, S. Lafortune, and D. Teneketzis, Coordinated decentralized protocols for failure diagnosis of discrete event systems, Journal of Discrete Event Dynamical Systems: Theory and Application, 10(12), (2002).
[37] T. Yoo and S. Lafortune, Polynomial-time verification of diagnosability of partially-observed discrete-event systems, IEEE Transactions of Automatic Control, 47(9), 14911495, (2002).
[38] Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., Vernotte, A.: Chapter 2 - recent advances in model-based testing. Advances in Computers 101,53120 (2016), http://dx.doi.org/10.1016/bs.adcom.2015.11.004.
[39] Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches.STVR 22(5), 297312 (2012), http://dx.doi.org/10.1002/stvr.456
[40] Incki, K., Ari, I., Sozer, H.: A survey of software testing in the cloud. In: 6th IEEE International Conference SERE-C. pp. 1823 (June 2012).
[41] L. Cacciari and O. Rafiq: Controllability and Observability in Distributed Testing, Information and Software Technology, (1999).
[42] Khoumsi Ahmed: A Temporal Approach for Testing Distributed Systems, IEEE Transactions on Software Engineering, vol. 28, 1085-1103, (2002).
[43] Hierons, M., Merayo, G., Núñez, Manuel: Timed implementation relations for the distributed test architecture, Distributed Computing, Vol. 27, (2014).

Appendix 3

III

Pal, Deepak; Vain, Jüri (2019). A systematic approach on model refinement and regression testing of real-time distributed systems. 9th IFAC Conference on Manufacturing Modelling, Management and Control, MIM 2019 : Berlin, Germany, 28-30 August 2019, Proceedings. Ed. Ivanov, Dmitry; Dolgui, Alexandre; Yalaoui, Farouk. Elsevier, 1091-1096

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnline CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 52-13 (2019) 1091–1096

A Systematic Approach on Modeling Refinement and Regression Testing of Real-Time Distributed Systems

Deepak Pal * **Jüri Vain** *

* *Tallinn University of Technology, Tallinn, Estonia (e-mail: {deepak.pal, juri.vain}@taltech.ee).*

Abstract: Industry 4.0 aims at highly flexible and digitized model of industrial production. This requires vertical integration of different operations in the manufacturing process to promote reconfigurability and better collaboration between distributed components (*Modularity & Integrability*) and to handle growing complexity in manufacturing systems. On the other hand, the flexibility of reconfiguration leads to new challenges for the testing process (*i.e Diagnosability*). It has now become a priority to quality assurance team to ensure that reconfiguration did not introduce any new faults, often referred to as Regression Testing. In this paper, we present a testing framework for real-time distributed manufacturing systems to verify reconfiguration correctness and to generate tests for its implementation testing. We present a systematic modeling refinement approach where several refinements (reconfigurations) can be modelled separately from an abstract model and composed with it. We use the reconfiguration model derived by refinements to demonstrate distributed regression testing by partitioning it into a set of distributed communicating testers making model based testing for manufacturing automation systems scalable.

Keywords: Reconfigurable Manufacturing System, Regression Testing, Distributed Systems, Model Based Testing

1. INTRODUCTION

The concept of reconfigurable manufacturing automation systems (MAS) has been used to address the challenges of dynamic and unpredictable events W. Shen (1999). Reconfiguration process enables a system to transform from one configuration to another, improving the efficiency of the system in response to the unpredictable events. The reconfiguration of the complete manufacturing process lead-time is known as ramp-up period, i.e., the time to design and develop or reconfigure the manufacturing process, and to ramp-up to high-quality production, full volume to meet market demand. The Ramp-up period has become the bottleneck of any manufacturing process because it requires systematic testing (*i.e Diagnosability*) before the system goes live. Industry 4.0, manufacturing systems operate with real-time field data and have grown to the scale of global geographical distribution with numerous services and applications forming an ubiquitous computing network. Further, the heterogeneity of the environment, where possibly millions of sensors, actuators, and different smart devices in conjunction with intelligent software form a complex system and introduce a new dimension to real-time testing pose significant challenges.

Along with the flexibility to configuration changes of MAS grows the challenge to ensure that reconfiguration did not introduce any new faults, often referred to as *Regression Testing*. The need for automated and rigorous online testing have given rise to the use of Model-Based Testing (MBT) and the development of several commercial and academic MBT tools. For instance, smart connected factories with Internet of things (IoT) based control systems are a new technology in manufacturing industries which undergoes frequent change in requirement specifications and tools. This expects reduction in testing efforts and costs. In this context, MBT offers an automated tool support for regression testing and platform independence, thus aiming to lower the testing effort of IoT Richter (2014). We interpret MBT in the standard way, i.e. as conformance testing that compares the expected behaviors described by the system model with the observed behaviors of an actual implementation.

We review related work from two perspectives, first we discuss related work on methodology that supports test model construction by refinements and their verification. Second we review related work on distributed regressing testing for real-time manufacturing systems. F.Long (2016) present an approach for modelling the production systems in industry 4.0 and their availability using high-level Petri nets ECSPN but did not address the verification of them. K. Sarna (2012) present a refinement-based aspect oriented modelling approach for construction of SUT models and their verification using Uppaal time-automata and model checker. This paper explains the construction of different aspect models and provides: off-line generation by model checking, and tron based for on-line. Zech et al. (2017) present a method for model-based regression testing, based on the model versioning engine MoVE and

* This research was supported by the Estonian Ministry of Education and Research institutional research grant no. IUT33-13.

10.1016/j.ifacol.2019.11.341

additional Object Constraint Language (OCL) statements to implement different test case selection, reduction, and prioritization strategies. This paper explains the process of generation and selection of regression test cases from the UML basic behavioral models which is not applicable for distributed real-time systems. Similarly, S. Ulewicz an approach is presented where monolithic tester is supported in choosing, prioritizing and performing relevant test cases during system regression testing for non real-time systems. According to our best knowledge the regressing testing for real-time distributed systems have not been presented systematically where models of system and their refinements are constructed automatically in Uppaal timed automata formalism yet.

In model-based software engineering the development processes are based on system models. These models describe a system from different viewpoints and on different levels of detail. Due to the multitude of viewpoint models their composition is inevitable in the system integration phase where the models need to be checked for consistency and integrity of crucial design aspects. In this paper, a testing framework for real-time distributed manufacturing systems is presented, applicable from the model refinements with new specifications to performing model-based distributed regressing testing. We present a systematic model refinement approach where several refinements (reconfigurations) can be modelled separately from an abstract model and composed with it to integrate these refinements. We use the same refined system model for distributed regression testing by partitioning it into a set of distributed communicating testers, partitioning algorithm explained in D. Pal (2018).

2. PRELIMINARIES

2.1 Model Based Distributed Testing

Model based regression testing is a form of selective retesting of the existing system or component to ensure its behaviour is same and still meet the requirements after modifications. The complexity growth of software has made the code-based regression testing ineffective, expensive and a time-consuming task. The MBT is used to derive test cases from software models under test to enable early detection of any requirements violation.

Modelling Implementations of Real-Time Systems To define our testing architecture formally we need to introduce a semantic foundation for modelling the real-time systems. We describe the notions of timed input output automata (TIOA) introduced by R. Alur (1994); J. Bengtsson (2004) as a formalism to model the behavior of real-time systems over time. We consider time domain $\mathbb{T}$ as set $\mathbb{R}_{\geq 0}$ of non-negative reals called clocks (delays) and Σ as a finite set of actions. For the formal syntax and semantics of TIOA we refer the reader to J. Bengtsson (2004). *Example 1:* Consider the TIOA specification $\mathcal{S}pec$ shown in Fig 1 (a), $in[1]?$, $in[2]?$, $in[3]?$ denotes the input to the system and $out[1]!$, $out[2]!$, $out[3]!$ denotes the output produced in response to input to the system. The timed $\mathcal{S}pec$ can be expressed in similar language as follow: exactly at 5 time units after the system received the input $in[1]?$ it produces either output $out[3]!$ exactly at 2 time units or, failing to do that, output $out[1]!$ exactly at 3 time units. The clock cl is set to 0 when passing the transition. A timed trace ρ is a sequence of timed-stamped input/output actions followed by a delay, e. g. $\rho_{Seq(\mathcal{R})} = (5{\cdot}in[1]!){\cdot}\ (8{\cdot}out[1]?){\cdot}\ (15{\cdot}in[2]!){\cdot}\ (18{\cdot}out[2]?){\cdot}0$. We have $\mathcal{S}pec$ After $(5{\cdot}in[1]!){\cdot}0 = \{(l_1, 0)\}$, $\mathcal{S}pec$ After $(5 \cdot in[1]!) \cdot (8 \cdot out[1]?).0 = \{(l_5, 0)\}$, $Out(\mathcal{S}pec$ After $(5 \cdot in[1]!) \cdot (7 \cdot out[1]?).0) = \mathbb{T}$, $Out($ $\mathcal{S}pec$ After $(5 \cdot in[1]!) \cdot (8 \cdot out[1]?).15) = \{in[2]!\} \cup \mathbb{T}$.

Uppaal Timed Automata (UTA): In our approach UTA G. Behrmann (2004); J. Bengtsson (2004) are used as a formalism to illustrate TIOA to model SUT behavior. This choice is motivated by the need to test the SUT with timing constraints so that the impact of propagation delays between the SUT and the tester can be taken explicitly into account when the test cases are generated and executed. UPPAAL is based on the definition of timed automata, which is introduced by R. Alur (1994). For the full formal syntax and semantics of UTA we refer to G. Behrmann (2004); J. Bengtsson (2004).

Modelling distributed n-Ports: We model a multi-ports TIOA in UTA by splitting the transition with multiple communication actions to a sequence of transitions each labeled with exactly one I/O-action and connected via committed locations, so that all ports of such group are updated instantaneously in the order they are specified in the tuple. In Fig. 2 the labels on the transition represent the i/o actions and the transition tuple (l_0, l', $in[1]!$ /($out[1]!$, $out[3]!$)) represents the sequence of transitions each labeled with exactly one action and connected via committed locations, l_0 represents the *idle*, and l' represents the *Done* location. Let P_{l_n} denotes a set of ports accessible in the physical location l_n where $n \in \mathbb{N}$; I is a n-tuple (I_1, I_2,...,I_n), where I_i is the finite set of inputs at port i, $I_i \cap I_j = \phi$ for $i \neq j$ and $i,j = 1,...n \in \mathbb{N}$. Similarly, O is a n-tuple (O_1, O_2,..., O_n), where O_i is finite set of outputs at port i, $O_i \cap O_j = \phi$ for $i \neq j$ and $i,j = 1,...n \in \mathbb{N}$. Each port may receive outputs of other port, i.e $O = (O_1 \cup \{\varepsilon\}) \times (O_2 \cup \{\varepsilon\}) \times ... \times (O_n \cup \{\varepsilon\})$, here $\{\varepsilon\}$ denotes the empty output in response to input to $\mathcal{SUT}$.

2.2 Timed Input/Output Conformance Relation

In order to define the conformance relation, we recall the timed input/output conformance relation (`tioco`) introduced in M. Krichen (2004); M. Mikucionis (2004). They propose extension of `ioco` relation with timing constraints including clock valuations with the set of observable actions. The communication between the specification and the SUT involves controllable inputs of the SUT and observable outputs of the SUT. In this work, we introduce the `ioco` at first as defined by Tretmans (1996). The behaviour of `ioco`-correct implementation should respect after some observations following restrictions: *(i)* the out-

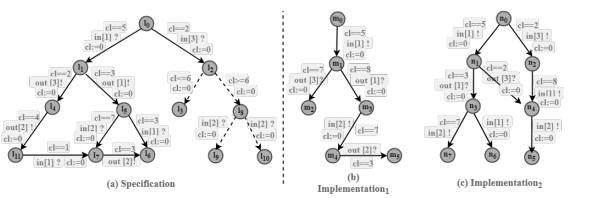


Fig. 1. Models of TIOA specification and implementations

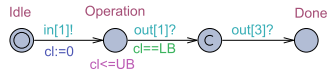


Fig. 2. Modelling pattern of multiport timed automata

puts produced by SUT should be the same as allowed in the requirements model; *(ii)* if a quiescent state (a situation where the system cannot evolve without an input from the environment) is reached in SUT, this should also be the case in the model; *(iii)* any time an input is possible in the model, this should also be the case in the SUT. In addition to `ioco`, `tioco` introduces the time delays observable on test interface. This is explained by means of following example (for detailed definition of (`tioco`) we refer to M. Krichen (2004); M. Mikucionis (2004). *Example 2:* Consider the timed I/O automata specification $\mathcal{S}pec$ and implementations $\mathcal{I}mpl_1, \mathcal{I}mpl_2$ shown in Fig. 1. Based on `tioco` relation, we can verify that if $\mathcal{I}mpl_1$ conforms to $\mathcal{S}pec$, for example: $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]?)) = \mathbb{T}$ and $Out(\mathcal{I}mpl_1 \text{ After } (5 \cdot in[1]!)) = \mathbb{T}$; $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]?) \cdot 8) = \{out[1]!\} \cup \mathbb{T}$ and $Out(\mathcal{I}mpl_1 \text{ After } (5 \cdot in[1]!) \cdot 8) = \{out[1]?\} \cup \mathbb{T}$; $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]?) \cdot 7) = \{out[3]!\} \cup \mathbb{T}$ and $Out(\mathcal{I}mpl_1 \text{ After } (5 \cdot in[1]!) \cdot 7) = \{out[3]?\} \cup \mathbb{T}$, proves that $\mathcal{I}mpl_1$ `tioco` $\mathcal{S}pec$. Similarly, we can prove that $\mathcal{I}mpl_2$ ~~`tioco`~~ $\mathcal{S}pec$ i.e. $Out(\mathcal{S}pec \text{ After } (5 \cdot in[1]?) \cdot (7 \cdot out[3]!)) = \mathbb{T}$ and $Out(\mathcal{I}mpl_2 \text{ After } (5 \cdot in[1]!) \cdot (7 \cdot out[3]?)) = \{in[2]!\} \cup \mathbb{T}$; $Out(\ \mathcal{S}pec \text{ After } (5 \cdot in[1]?) \cdot (8 \cdot out[1]!).18) = \{out[2]!\} \cup \mathbb{T}$ and $Out(\mathcal{I}mpl_2 \text{ After } (5 \cdot in[1]!) \cdot (8 \cdot out[1]?).18) = -$.

3. MODEL REFINEMENT AND VERIFICATION

In this section we present the model refinement approach based on abstraction model. We consider that system reconfiguration starts from a base model ($\mathcal{M}_b$) which already exists and is modelled based on initial requirements. As shown in Figure 3, for systematic specification and verification of its reconfigurations an abstract model ($\mathcal{M}_a$) is constructed from a base model and verified to prove that abstraction preserves important properties (safety and liveness) ϕ of the base model i.e. $\mathcal{M}_b \models \phi$ implies $\mathcal{M}_a \models \phi$. Then an abstract model can be refined as per the new requirements. Such refinements, are conservative if they guarantee that properties ϕ, which were verified for an abstract model, still hold for the refinement. It is important to mention here that abstract models are constructed in a way that they can be re-used in a different context, even for different systems in a process.

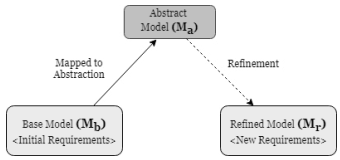


Fig. 3. Refinement Approach

From the modelling point of view the abstraction based refinement requires just a *place holder* element in the abstract model which defines where the substitution is applied. The reconfiguration is implemented then as K. Sarna (2012). We call these refinement operators location refinement (denoted by $\sqsubseteq_l$) and edge refinement (denoted by $\sqsubseteq_e$) shown in Figure 5.

Let the refinement $\mathcal{M}_r$ that specifies the reconfigured fragment of a system be composed by synchronous parallel composition $\|_{sync}$, so that, $\mathcal{M}_a \sqsubseteq \mathcal{M}_a \|_{sync} \mathcal{M}_r$ where $\sqsubseteq \in \{\sqsubseteq_l, \sqsubseteq_e\}$. Synchronous composition of $\mathcal{M}_a$ and $\mathcal{M}_r$ should preserve the semantics of $\mathcal{M}_a$ also after superposition. Technically, this composition means that entry and exit points of the refined $\mathcal{M}_r$ have to be synchronized (via auxiliary channel) with an edge in case of edge refinement or before and after edges in case of location refinement of $\mathcal{M}_a$. *Example 3: Distributed manufacturing automa-*

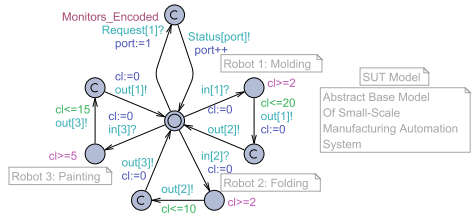


Fig. 4. Abstract Model of Robots in RAMS

tion system presented in Figure 4, specifies the abstract model of functional behavior of three robots distributed at different location. The automata specify a simplified manufacturing process where *Robot 1* is making *work-piece* by shaping liquid or pliable raw material using a rigid frame called a mold. Then final *work-piece* is sent to *Robot 2* for folding and similarly after folding the *work-piece* is passed to *Robot 3* for painting. We consider above example as real-time distributed system where each *Robots* reside in specific department and synchronize motion and tasks with others through signals or directly with smart sensors. The abstract model shown in figure has 3 ports (p_1, p_2, p_3) in geographically different places with inputs/outputs $in[1]/out[1]$, $in[2]/out[2]$ and $in[3]/out[3]$ at ports $p1$, p_2 and p_3 respectively. The UTA models defines the expected global behavior of any potential SUT. Each expected global behavior is expressed as the sequence of labels of UTA model edges.

As per technical specification $\mathcal{S}pec$, *Robot 1* receives the request from controller with input $in[1]?$ and produce the $out[1]!$ at port 1 and $out[2]!$ at port 2. We assume that each *Robot* generates two kind of outputs, first to its local port which can be depicted as the final *work-piece* or as output and second output to following *Robot* for synchronization. After the molding process *Robot 1* synchronize with *Robot 2* by sending $out[2]!$ at port 2. Similarly, *Robot 2* after folding process synchronize with *Robot 3* by sending $out[3]!$ to *Robot 3* at port 3. We call this baseline abstract model as version 1.0 before adding more functionality to it and upgrading to version 1.1. For illustrating any modifications/reconfigurations in MAS, we provide examples of two refinements to abstract *Robot 1* behavior, depicted in Figure 6. It shows location and edge refinement to abstract model. In addition to abstract model depicted in Figure 4, there is a new requirement to the process, *milling* has to be done to *work-piece* (by *Robot 1* department) to specific dimension after molding it. To implement new changes

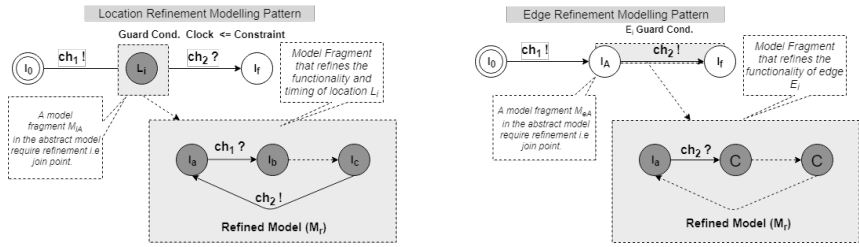


Fig. 5. Location and Edge Refinement Modelling Pattern

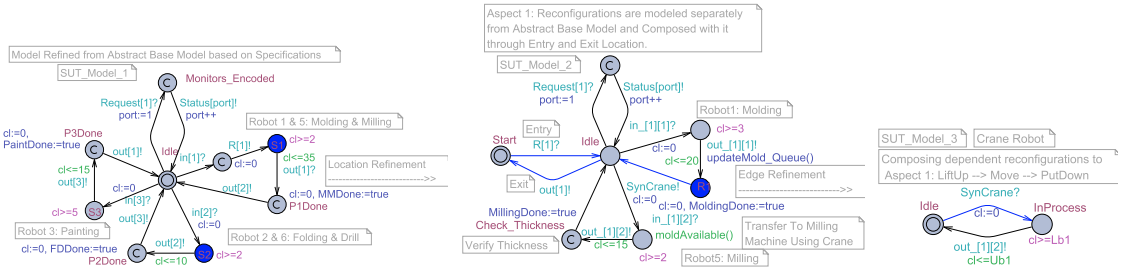


Fig. 6. Refinement Models After Reconfigurations

to existing configured manufacturing automation system, we reconfigured the *Robot 1* process and integrated a new *Robot 5* for milling process and *Robot 7* for Crane as shown in Figure 6. Also location *S1* in blue color is modified and assumed to be refined by location refinement to extend the functionality. Similarly, outgoing edge in refined model is considered edge refinement to integrate the *Crane Robot* into the process. Similarly, requirement for *Drill* process can be configured into existing manufacturing system.

3.1 Refinement Verification

The interaction of *Robot 1* and *Robot 5* is safety-critical, because both robots belong to same work bench and are coordinating with *Crane Robot*. They may come into collision state because of the control error that can create erratic behavior to *Crane Robot* or unauthorized access to workbench; improper installation of *Robot 5* etc. We use the model checking to prove that, first, the refined model fulfills the liveness and safety properties which ensure the correct behavior and, later, verifying functional behavior of robots. We formalize the properties using timed computation tree logic (TCTL) and verify them using Uppaal model checker. In TCTL, the properties are formalized as follows:

- Property 1. *Interference Free New Updates*:
  No variable of abstract model is updated in refined model i.e. there is only one function *updateMold_Queue()* in refined model that has variable but its scope is local and does not affect any functionality in the abstract model. Hence, property 1 is satisfied and this allows an interference freedom with the abstract model;
- Property 2. *Preservation of Non-Blocking*:
  No deadlock in the model, the moment parallel synchronous execution triggers abstract model via channel *in[1]!*, the refined model always returns to *Start* location within specified time i.e. 35 time units. This property can be verified by two ways:
  1. A[] no deadlock;
  2. A[] MoldingDone ==True and MillingDone == True and (SUT_Model_2.Start imply SUT_Model_1.p1.Done)
- Property 3. *Non-Divergence*:
  invariant(L_i) $\equiv$ x $\leq$ n for all clocks x $\in$ $Clock_{\mathcal{M}_a}$, n $< \infty \Rightarrow \exists$ d $\leq$ n: $[\mathcal{M}_r, l_0 \vDash l_0 \rightsquigarrow_d l_f]$. The property holds in UTA models, since *SUT_Model_2* (refined model) always returns control to abstract model *SUT_Model_1* atmost 35 time units while the join point carrier location *S1* has invariant $cl \leq 35$ which yields that for non-divergence the condition *SUT_Model_2*.cl $\leq$ *SUT_Model_1*.cl must be satisfied.

Thus, alternative configurations (original and its reconfiguration) are reflected in the models as different refinements of the common abstract system model. If the refinement correctness conditions Property 1 to Property 3 are satisfied it guarantees that the properties verified for an abstract model are preserved and all reconfiguration verification reduces to verifying local properties Property 1 to Property 3 only. In our example, the refined automata have to preserve above properties and if the refined model conforms to abstract model, then the given refinements in Figure 6 are correct. Verifying refinement conditions for correctness guarantees a correct refinement. Such a refinement approach makes formal verification of real-time distributed systems more scalable and their reconfiguration provably correct.

4. DISTRIBUTED AUTOMATED REGRESSION TESTING

In D. Pal (2018), we proposed a distributed test framework for testing real-time distributed systems augmented

with monitors, where online monitors are used to record relevant events (timing and order of input/output events at test interface ports). Online monitored data is used to obtain a coherent view of the system and to simplify distributed testing, where local testers synchronize with each other via communicating with these monitor. We showed that the proposed test architecture is test reaction time wise more scalable than centralized remote test architecture for testing large number of geographical locations (ports) in a system, architecture depicted in Figure 7.

Given a centralized remote tester model we apply a partitioning algorithm to map it to a set of distributed testers. We prove that the proposed approach not only preserves the functional correctness of the centralized remote testers but also satisfies stronger timing constraints needed for solving distributed test controllability and observability issues. In the next subsections, we show the distributed regression testing steps for distributed system.

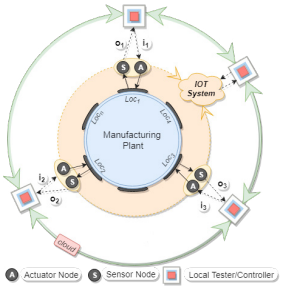


Fig. 7. Distributed Test Architecture

4.1 Baseline Tester Model Version 1.0

For generating distributed local testers for performing actual regression testing, it is require to have existing baseline tester model of stable version of SUT. According to MBT taxonomy, such baseline tester models are often constructed by MBT approaches. In our case, the full monolithic centralized tester model (baseline test suite) for stable version 1.0 (abstract model) shown in Figure 4 is generated by model checking queries.

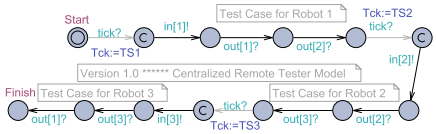


Fig. 8. Centralized Tester of SUT Model Version 1.0

4.2 Generating Distributed Local Testers Version 1.0

An alternative to the centralized tester is distributed testing the architecture of which is shown in Figure 7. Here the centralized tester model deployed on a centralized testing architecture is decomposed into a set of communicating (via SUT) distributed local testers using partitioning algorithm proposed in D. Pal (2018), one for each localized interface of the system. The generated distributed local testers shown in Figure 9, for more reading about distributed local testers communication and optimization over centralized tester refer to our work D. Pal (2018).

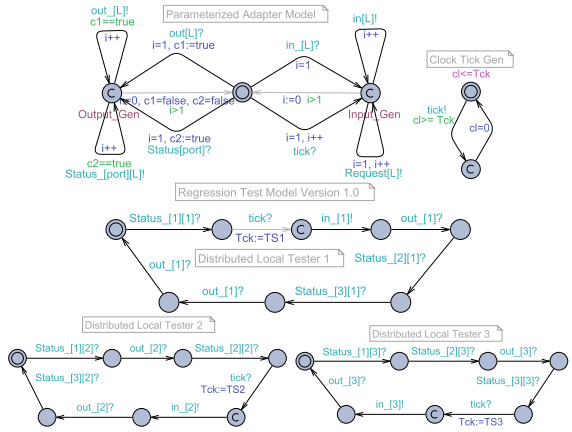


Fig. 9. Distributed Tester Models Version 1.0

4.3 Reconfigurations Identification and Test Selection

Identification of modifications in the stable version of system is the first requirement of performing Regressions. This aims to obtain any modifications in the abstract model by comparing the baseline test models and refined models along with actions, clock constraints and invariants. After identifying new changes and their impact on existing component, the next step is to retesting the systems with selective test cases for example test cases targeted to modified parts of the system. In case of new functionality being added, selective tests can verify that the new features work as per the requirement and design specifications while regression testing can show that the new code has not broken any existing functionality.

4.4 Maintenance of Broken Regression Tests

Regression packages often contain tests that cover the core functionality that will stay the same throughout the evolution of the system. A lot of the old test cases may become inapplicable as basic functionalities may have been replaced and removed by new functionality. Therefore, the regression packages require maintenance regularly to reflect changes to the application. For each new reconfiguration to the system, new test cases needs to be develop that becomes the part of regression to be executed after the reconfiguration is deployed.

New requirements and modifications applied to model and upgraded to version 1.1 are shown in Figure 6. Abstract SUT model version 1.0 is extended by adding new functionality to *Robot 1 and Robot 2*. As said earlier, baseline test suites may become inapplicable as basic functionality of *Robot 1 and Robot 2* replaced with new requirements. Therefore, it is require to upgrade the baseline tester model with new tests for further testing purpose. As per requirements from the market, *milling* has to be done to *work-piece* (by *Robot 1* department) to specific dimension after molding it. To implement new changes to existing

configured manufacturing automation system, we reconfigured the *Robot 1* process and integrated a new *Robot 5* for milling process and *Robot 7* for Crane as shown in Figure 6. Similarly, requirement for *Drill* process can be configured into existing manufacturing system.

For the simplicity, we show the refinement for milling process only. After identifying new changes in refined model version 1.1, we classified the test cases for modified parts of the system, i.e test case for *Location 1* only where we enhanced the functionality of *Robot 1* by integrating *Robot 5 and 7*. Since new added functionality does not have any side-affect on other part of the model, we modified only part of centralized tester model, i.e test case for *Robot 1* and other parts of model, i.e. test case for *Robot 2 and 3* do not require any update. A new centralized tester model for location 1 generated from refined model is shown in Figure 10 and their corresponding distributed local tester models shown in Figure 11. Note: In our approach distribution algorithm applied only on refinement model, instead of developing and partitioning a full centralized model from the scratch. We optimized the regression tests generation for model version 1.1 by identifying changes and selectively refined the part of abstract model.

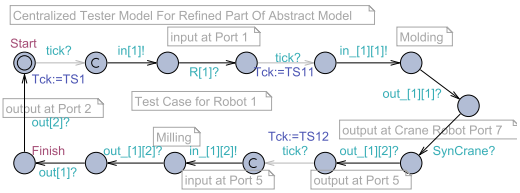


Fig. 10. Centralized Tester Updated for Robot 1

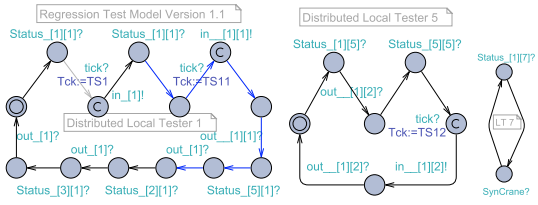


Fig. 11. Distributed Tester Models Version 1.1

4.5 Regression Test Execution And Analysis

This step requires the execution of new regression test package against new configuration deployed and its analyses to detect any regression bug/defects. There are two major possibilities if some bug/defects are discovered in the test results. First automated regression test models may have logical error. Alternatively, there is a real bug found in the system which needs to be corrected and deployed or rolled back to previous version depending on the priority and time required to fix it.

5. CONCLUSION

With the growing unpredictable events MAS are made flexible and reconfigurable. As systems in manufacturing automation are becoming more complex, it has become a high-priority to QA team to ensure that reconfiguration will not introduce any faults, often referred to as Regression Testing. The need for automated online testing and system correctness assurance have given rise to the use of MBT and the development of several commercial and academic MBT tools. In this paper, a testing framework for real-time distributed manufacturing systems is presented to verify reconfiguration correctness and to generate test for its implementation testing. We presented MAS reconfiguration steps abstractly as its refinements. We used the reconfiguration model derived by refinements and verified also for test generation. We demonstrated distributed regression testing by partitioning it into a set of distributed communicating testers making model based testing for MAS scalable. In future work, we have plan to do empirical research, i.e., controlled experiments and industrial case studies to investigate the impact of the presented approach on time to market improvement of time critical MAS.

REFERENCES

Pal, D., Vain, J. (2018). Model based approach for testing: Distributed real-time systems augmented with online monitors. In *Databases and Information Systems*, 142–157. Springer International Publishing, Cham.

Long, F., Zeiler, P., Bertsche, B. (2016). Modelling the production systems in industry 4.0 and their availability with high-level petri nets. *IFAC*, 145-150.

Behrmann, G., David, A., Larsen, K.G (2004). A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems., TNCS*, 200–236, vol. 3185.

Bengtsson, J., Wang, Y. (2004). Timed automata: Semantics, algorithms and tools. In *Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, 87–124.

Sarna, K., Vain, J. (2012). Exploiting aspects in model-based testing. *FOAL12: 11th Workshop on Foundations of Aspect-Oriented Languages.*, 1(2), 45–47.

Krichen, M., Tripakis, S. (2004). Black-box conformance testing for real-time systems. *In: 11th International SPIN Workshop on Model Checking Software.*, 109–126.

Mikucionis, M., Larsen, K.G., Nielsen, B. (2004). T-uppaal: Online model-based testing of real-time systems. *In: 19th IEEE ASE 2019*, 396–397.

Alur, R., Dill, D.L. (1994). In *A theory of timed automata.* Theoretical Comp. Science.

Richter, R. (2014). Does the internet of things force us to rethink our test strategies?

Ulewicz, S., Vogel-Heuser, B., Simon, H., Bohlender, D., Obster, M., Kowalewski, S. (2017). In *A priori test coverage estimation for automated production systems: Using generated behavior models for coverage calculation. ETFA*, 1-4.

Tretmans, J. (1996). Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 103–120.

W. Shen, D.N. (1999). Agent-based systems for intelligent manufacturing: A state-of-the-art survey. *Knowl Inf Syst*, 1(2), 129–56.

Zech, P., Kalb, P., Felderer, M., Atkinson, C., and Breu, R. (2017). Model-based regression testing by ocl. *International Journal on Software Tools for Technology Transfer*, 19(1), 115–131.

Appendix 4

IV

Pal, Deepak; Vain Jüri; Srinivasan, Seshadhri; Ramaswamy, Srini (2017). Model-based maintenance scheduling in flexible modular automation systems. 22nd IEEE International Conference on Emerging Technologies and Factory Automation, EFTA' 2017 : September 12-15, 2017, Limassol, Cyprus, 1-6

Model-Based Maintenance Scheduling In Flexible Modular Automation Systems

Deepak Pal*, Jüri Vain*, Seshadhri Srinivasan†, Srini Ramaswamy‡
*Department of Computer Science, Tallinn University of Technology, Tallinn, Estonia
Email: {deepak.pal, juri.vain}@ttu.ee
†Berkeley Education Alliance for Research, Singapore
Email: seshadhri.srinivasan-bears@berkeley.sg
‡ABB Inc., USA
Email: srini@ieee.org

***Abstract*—Industry 4.0 aims at highly flexible and digitized model of industrial production that is smarter and more reliable than the current possibilities. This requires vertical integration of different operations in a manufacturing to promote reconfigurable smart factory. This investigation proposes a method to schedule maintenance operations using formal methods considering power balance and production constraints in process industries. First, we provide an optimization model for scheduling maintenance and operation along production schedules. Second, we use timed model checking to schedule maintenance considering various physical and operating constraints. Third, we illustrate the method in a smart aluminium factory. The main contribution of this investigation is the integration of optimization models and formal methods in one framework, which leads to verified production/maintenance schedules and contributes to the objectives of Industry 4.0.**

I. Introduction

The enterprise wide optimization (EWO) is an emerging concept that is focused on using optimization across different layers of manufacturing process to obtain holistic production, procurement and resource management schedules. The use of EWO to steel, motor manufacturing and other process industries have been studied widely [1]. In the literature, the maintenance operations are solved using off-line optimization models which do not reflect emerging maintenance requirements of a process industry. Similarly, the power-balance constraints are not considered, this is important as more renewables are getting integrated in industries. A common feature in these investigations is that the optimization routines lead to either mixed integer linear program (MILP) or a non-linear optimization problems. While MILP problems are NP-hard, a solution can be obtained using relaxations or branch-and-bound techniques. On the other hand, non-linear optimization problems involving binary decisions variables are hard to solve. In such problems obtaining an initial guess that is feasible is also a NP-hard problem. Moreover, global optimal solution is seldom feasible. Consequently, relaxed versions of the problem are widely employed that is rather difficult to model the real-scenarios. Therefore, the use of heuristic optimization techniques such as genetic algorithm, particle swarm optimization, or other techniques are usually employed. Contrastingly these are off-line optimization tools and applying them to solve a scheduling problem on-the-fly is rather difficult due to absence of convergence properties. Therefore new techniques for solving optimization problems involving binary and real-variables are required. More importantly, such solutions needs to be verified not to violate physical and operating constraints, even with feasible solutions for them to be applied in EWO.

978-1-5090-6505-9/17/$31.00

In past, models checkers have been applied in solving combinatorial optimization problems in which one best combination of values is selected as solution from a set of given values e.g. [2] - [6]. The problem of scheduling processes have been considered using a range of available model checkers, e.g. Uppaal Cora [7]. Authors in [8] made use of the model checker SPIN to solve scheduling problems, using language PROMELA. In [9] author proposed the model of timed automata, as a model for posing and solving time-dependent planning and scheduling problems. Most recently, [12] used price time automata model and uppal cora to solve scheduling problems in steel industry. Here too the maintenance operations were not considered in their analysis. However, the well-known state-space explosion problem may arise when the model become too complex and existing sequential model checkers such as (Uppaal, SPIN) fail to provide parallelization and more computation power, which may restrict their use on industrial scale. To fill this gap, we propose an approach to find optimal schedule using distributed DiVinE model checker and algorithm in [13].

This investigation presents a method to optimize the production schedules in an aluminium industry along with maintenance and power balance constraints. First, we model the production process pertinent for solving a nonlinear optimization problem involving binary and real–variables. The formulation includes maintenance schedules and power balance constraints. Second, we provide an alternative solution using formal methods to check the feasibility of the solution obtained using optimization models. The feasibility check verifies the validity of optimization solution under given maintenance and power balance constraints. We also demonstrate how the manufacturing and maintenance schedules can be extracted from the fastest execution traces of the formal model which constitutes a time optimal schedule using DiVinE explained in [13]. Finally, we illustrate the combined use of optimization models and formal methods in an aluminium industry.

The paper is organized as follows- Section II introduces the different manufacturing steps in a prototypical process industry, aluminium in our case. The optimization model and its complexity are analysed in Section III. The formal method to solve the scheduling problem using time automata models and reachability analysis on them is presented in Section IV. Analysis and model execution traces as plant schedule is discussed in Section V. Finally, we draw conclusions.

II. Problem Statement

A. Plant Description

The conventional production process of aluminium rolled strips is schematically categorized into three main steps and maintenance

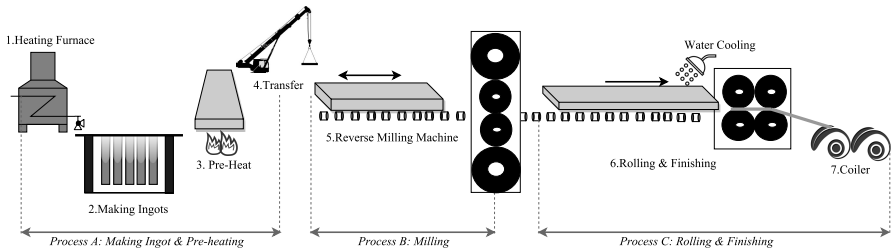


Fig. 1: Conventional Process for Aluminum Strip

as shown in Fig. 1 and described in the following steps.
(i) In step 1–the aluminium is obtained from the input raw materials in heating furnace by the use of electricity. The alluminium obtained in molten form transforms into ingots or slabs. The ingots are transferred to refining furnace where they are pre-heated to the melting point, which removes the impurities and carbon contents from ingots. Thereafter, solid ingots are transported by a crane to a reverse milling machine.
(ii) In step 2–the ingot is fed to a series of rolling stands consisting of work rollers and backup rollers. The work-rollers and backup rollers exert force on hot ingot to reduce its size to an aluminium strip of desired dimensions. The ingot is passed over rollers a number of times, where each stage changes its shape and dimension and it continues till desired product is achieved.
(iii) In step 3–the aluminium strip received is cooled by water spray and sent to the finishing mill where it is made as per the customer requirements. It undergoes surface finish to give desired mechanical properties and coating providing a thin strip of alluminium. The thin strip is thousands of feet in length and fed to the coiler to form the final product in the form of aluminium coil.

TABLE I: Energy Consumption Description

Process	Comments
A (Making Ingots & Pre-heat)	*Energy intensive and its output feeds the process B, C.*
B (Milling)	*Requires input from process A and is followed by process C. The process consumes less energy than A and C, but its energy consumption is more than process D.*
C (Rolling & Finishing)	*Process C requires input from B. The energy consumption is less than A, but greater than B and D.*
D (Maintenance)	*Maintenance operation needs to be planned between these tasks for the component with possible failure information*

Scheduling aluminium coil production is a complex multistage process involving interdependent systems where both information and decision-making are logically and spatially distributed with distinct objectives and constraints, for example job scheduling and production constraints, power consumption constraints, set up and production costs. In this paper, the scheduling problem deals with the production and planning in a typical milling and rolling machines, wherein rolls of aluminium are processed. In the rolling mill as shown in figure 1, the preparation of ingots and pre-heating is carried out in process A, followed by the milling process shown in B, and the final step of rolling and finishing in process C. Lastly, the maintenance operations are based on predictive and emergency maintenance schedules that should intervene the production processes as little as possible.

III. Optimization Model for Aluminium Manufacturing Process

Nomenclature

Constants

T	Length of the planning horizon
$\mathcal{T} = \{1, ...T\}$	Set of Planning periods
N_i^{min}, N_i^{max}	Minimum and Maximum Production Levels in process i
$N_{i,t}^{max}$	Maximum number of units that can be produced in time-period,t
P_i^{min}, P_i^{max}	Minimum and Maximum Power Consumption in Unit i
$\mathcal{I}$	Set of tasks in the production
r_i^{up}, r_i^{down}	Ramp up and down rates of power consumption of production task i

Decision Variables

$y_{i,t}$	binary variable, is 1 if the operation i takes place during t
$N_{i,t}$	Production level of the operation i during the time period t
$P_{i,t}$	Power Consumed by the task i during the time period t

A. Production Constraints

Let $D_{i,t}$ and $R_{i,t}$ denote the demand and reserve of the number of products from the particular production task i during the time interval t. Considering $N_{i,t}$ to be the number of units produced, the production and demand constraints are given by

$$N_{i,t} \geq y_{i,t}(D_{i,t} + R_{i,t}) \quad \forall i \in \mathcal{I}, \forall t \in \mathcal{T} \tag{1}$$

Further, the following physical constraint is enforced to model the production capability of the machines

$$\sum_{i \in \mathcal{I}} N_i^{max} \geq y_{i,t}(D_{i,t} + R_{i,t}) \quad \forall t \in \mathcal{T} \tag{2}$$
$$\sum_{i \in \mathcal{I}} N_i^{min} \leq y_{i,t}(D_{i,t} + R_{i,t}) \quad \forall t \in \mathcal{T}$$

B. Power Balance Constraints

Let P_t^g, P_t^r and $P_{i,t}^d$ denote the power generated or bought, renewable generation in the production unit, and demand of the i the task at time t, respectively. Then we have the following power balance constraints

$$P_t^g + P_t^r - \sum_{i \in \mathcal{I}} P_{i,t}^d y_{i,t} = 0 \qquad \forall t \in \mathcal{T} \tag{3}$$

Let $y_{i,t}$ denote the variable modelling the operation of the production task i at time t, then the following constraints are forthcoming

$$P_{i,t}^{max} \leq y_{i,t} P_i^{max} \qquad \forall t \in \mathcal{T} \tag{4}$$

Including the ramp-up constraints, we have

$$P_{i,t}^{max} \leq P_{i,t-1} + y_{i,t-1} r_i^{up} + P_i^{max}(1 - y_{i,t-1}) \qquad \forall t \in \mathcal{T}, \forall i \in \mathcal{I} \tag{5}$$

C. Minimum up and down times of the process

When a production task is switched on(off), it must either remain on(off) for at least $T_i^{on}(T_i^{off})$ consecutive periods considering production economics. Constraints (6) model the on/off constraints.

$$\begin{aligned} y_{i,t} - y_{i,t-1} &\leq y_i(\theta^{up}) && \forall i \in \mathcal{I}, \forall t \in \mathcal{T} \quad (6) \\ y_{i,t-1} - y_{i,t} &\leq 1 - y_i(\theta^{down}) && \forall i \in \mathcal{I}, \forall t \in \mathcal{T} \end{aligned}$$

D. Objective Function

The main objective is to maximize the profits while reducing the operating cost. We assume that the production cost is linear in the number of units produced and the power tariff is organized into two parts: fixed cost C_1 and variable $C_2(t)$. The objective is to minimize the operating cost given by

$$\min_{y_{i,t}, P^d_{i,t}, R_{i,t}} \sum_{i \in \mathcal{I}} \sum_{t \in \mathcal{T}} C_P \times N_{i,t} + C_1 + C_2(t) \times P^d_{i,t} y_{i,t} \quad (7)$$

E. Optimization Model

The problem of reducing operating cost for a time-horizon $\mathcal{T}$ is solved by obtaining the forecasts on demand $D_{i,t}$, this includes also the forecasts for maintenance operations. The formulation above eliminates the need for additional binary variables. The receding horizon optimal control problem can be stated as:

$$\min_{y_{i,t}, P^d_{i,t}, R_{i,t}} \sum_{i \in \mathcal{I}} \sum_{t \in \mathcal{T}} C_P \times N_{i,t} + C_1 + C_2(t) \times P^d_{i,t}\ y_{i,t} \quad (8)$$

s. t.

$$\begin{aligned} & N^{max}_{i,t} \geq y_{i,t}(D_{i,t} + R_{i,t}) \quad \forall t \in \mathcal{T} \\ & \sum_{i \in \mathcal{I}} N^{min}_{i,t} \leq y_{i,t}(D_{i,t} + R_{i,t}) \quad \forall t \in \mathcal{T} \\ & P^g_t + P^r_t - \sum_{i \in \mathcal{I}} P^d_{i,t} y_{i,t} = 0 \qquad \forall t \in \mathcal{T} \\ & y_{i,t} P^{min}_i \leq P^d_{i,t} max \leq y_{i,t} P^{max}_i \\ & \qquad \forall t \in \mathcal{T}, \forall i \in \mathcal{I} \\ & P^{max}_{i,t} \leq P_{i,t-1} + y_{i,t-1} r^{up}_i + P^{max}_i (1 - y_{i,t-1}) \\ & \qquad \forall t \in \mathcal{T}, \quad \forall i \in \mathcal{I} \\ & y_{i,t} - y_{i,t-1} \leq y_i(\theta^{up}) \qquad \forall i \in \mathcal{I}, \forall t \in \mathcal{T} \\ & y_{i,t-1} - y_{i,t} \leq 1 - y_i(\theta^{down}) \qquad \forall i \in \mathcal{I}, \forall t \in \mathcal{T} \end{aligned}$$

The decision problem above has product of binary and real decision variables, thereby making it non-linear and non-convex [10]. Usually relaxations such as mixed logical dynamical constraints or big-M method are applied to solve the problem [11]. However, if maintenance requirements are added, then the problem becomes non-convex and nonlinear one. Greedy algorithms are usually used in this context to simplify the analysis. Albeit, finding a feasible solution with time-varying maintenance requirements for the problem by itself is NP-hard due to the bi-linearity among binary and real-valued variables. Further, guaranteeing feasibility is a challenging task. Therefore, new techniques for solving the optimization model that provides optimal yet feasible solution needs to be devised. Different from existing approaches, this investigation will use parametric model checking for solving the maintenance scheduling problem. While use of formal methods for solving scheduling optimization problem by itself is not new [12], the inclusion of time-varying maintenance schedules and applying non-convex optimization models have not been discussed in the literature to our best knowledge.

IV. Modelling of Manufacturing Systems with Maintenance Constraints

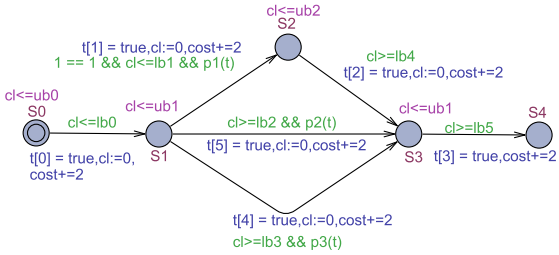
A. Modelling System and Verification with DIVINE

The Uppaal model checking tool is a sequential verification tool. It is treated as a benchmark among model checking tools. Uppaal timed automata (UTA) have proven to be a powerful formalism, to model and verify the behavior of real-time systems [14]-[16]. Timed automata introduced by [17] are widely used in different types of complex real-time systems, and used in scheduling control and model checking [18]. It provides a formalism to annotate a state transition system with timing constraints using finitely many real-valued clocks. The set of clock-variables track the time elapsed and can guard on transitions to restrict the behavior of automaton. A variety of optimization problems have been proven decidable for UTA including optimal scheduling, minimun-cost reachability [19], [20]. The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by an additional circle inside the location. Synchronous communication between the processes is by hand-shake synchronization links that are called *channels*. The interaction between processes as synchronizing actions allows the automata to make explicit the control flow of scheduling problems. The duration of the execution of the result is specified by the interval $[lb, ub]$ where the upper bound ub is given by the location *invariant* `cl<=ub`, and the lower bound lb by the *guard condition* `cl>=lb` of the transition. The *assignment* `cl:=0` on the transition ensures that the clock cl is reset when the transition is executed. For more reading about UPTA, its formal syntax and semantics, we refer to [19] [20].

Uppaal has a user friendly GUI and simulator which supports Computation Tree Logic (CTL), while DiVinE supports Linear Temporal Logic (LTL) for specification of temporal properties. DiVinE is a parallel verification tool based on DVE language. It's GUI provides an editor to code in DVE language and a simulator with step by step execution of model and a counter example. DiVinE's GUI does not provide features as Uppaal but DiVinE accept Uppaal xml format file as input to verify untimed properties. Moreover, its feature of parallel verification distinct it from all sequential verification tools. DiVinE is an explicit-state linear temporal logic (LTL) automata-based verification tool for reachability analysis of discrete distributed systems [21]. It employs an aggregate power to verify large systems models with better efficiency and memory usage. It is known that Uppaal verifies timed automata while DiVinE verifies untimed automata [22], [24]. In this paper, We extend the DiVinE capability of verifying the timed properties using guided model checking algorithm explained in [13]. The control guards for each transition are constructed by offline statistical analysis based on the given model and the test purposes. The generated control guards are labelled on transitions of the model and feed to DiVinE where controls are evaluated in every state when the selection between alternative outgoing transitions should be made. The execution of the model with control guards find the fastest and efficient path to goal state and provide a witness trace with optimal cost.

To run DiVinE and verify model properties, a LTL file is provided alongside the model xml file. LTL formula are loaded from this file and when a specific property is chosen, it is negated and a Büchi automaton is created. The Büchi automaton is multiplied with the timed automata on-the-fly to create a transition system on which the reachability analysis or accepting cycle detection algorithms are run [22], [23]. DiVinE runs the reachability analysis to find an error state or a time deadlocks. It performs universal verification only, i.e. it decides whether all runs meet given conditions. However, negation of existential formula is a universal formula. Therefore, the CTL specification in the form $A[]p$ or $A <> p$ can be directly translate into equivalent LTL formula Gp or Fp respectively. $E <> p$ can be translated to $G!p$ and $E[]p$ to $F!p$, but due to the mapping to universally quantified formuli the validation turns to falsification, i.e. the formulas in CTL and corresponding formula in LTL are not equivalent, in fact, they have exactly opposite meaning; hence when DiVinE says a formula $F!p$ does not hold in some model, Uppaal says that $E[]p$ is satisfied and vice versa. In the case of existential CTL formula that holds, DiVinE reports a counterexample to corresponding LTL formula, which is actually a witness for the original CTL formula [22].

B. From Diagnostic Traces to Plant Schedule




```
long m1(bool t[6]) {return (temp_LB1 * Tr_cost) + (lb4 * Tr_
long m2(bool t[6]) {return (lb2 * Tr_cost);}
long m3(bool t[6]) {return (lb3 * Tr_cost);}
long m4(bool t[6]) {return (lb4 * Tr_cost);}

bool p1(bool t[6]) {return m1(t)== min3(m1(t),m2(t),m3(t));}
bool p2(bool t[6]) {return m2(t)== min3(m1(t),m2(t),m3(t));}
bool p3(bool t[6]) {return m3(t)== min3(m1(t),m2(t),m3(t));}
```

Fig. 2: Sample Timed Automata with Control Guards

We demonstrate how a network of timed automata can be used for modelling the real time system (SUT) Since DiVinE does not support modelling formalism for real-time systems, We preferred Uppaal Timed Automata (UTA)[7] which have become the standard modelling language for real-time systems. Therefore we use UTA as the modelling language and DiVinE model checker as the reachability analysis tool.

Example 1. Consider a UTA in Fig. 2, here the decoration $p1(t)$, $p2(t)$ and , $p3(t)$ on transitions denotes the control guards, constructed by offline statistical analysis based on the given model and the test purposes [13]. The duration of the execution of the result is specified by the interval $[lb, ub]$ where the upper bound ub is given by the location *invariant* `cl<=ub`, and the lower bound lb by the *guard condition* `cl>=lb or cl<=lb` of the transition. The given values are $ub1 := 5$; $ub2 := 4$; $lb0 := 2$; $lb1 := 6$; $lb2 := 3$; $lb3 := 3$; $lb4 := 2$; $lb5 := 2$. A most common approach to the test generation is to formulate an informal set of test purposes transformed into some property of system such that model can be used to generate test cases for each property. A test purpose is specific property of system that tester wants to observe on the system under test. The test purpose can be directly formulated as a simple state reachability of *property* ($t[3] == true$) also known as single purpose test case generation. An example of test purpose (LTL property) of model in Figure 2 is expressed as:

```
#define Goal (t[3]==true)
#define Time (cl<=WatchDog)
#property G!(Goal && Time)
```

There are three possible paths (traces) to reach goal state:

- *Path 1 with cost = 10*: $S_0 \xrightarrow{t[0]} S_1 \xrightarrow{t[5]} S_3 \xrightarrow{t[3]} S_4$
- *Path 2 with cost = 10*: $S_0 \xrightarrow{t[0]} S_1 \xrightarrow{t[4]} S_3 \xrightarrow{t[3]} S_4$
- *Path 3 with cost = 8*: $S_0 \xrightarrow{t[0]} S_1 \xrightarrow{t[1]} S_2 \xrightarrow{t[2]} S_3 \xrightarrow{t[3]} S_4$

After modeling and encoding control guards to each transitions of timed automata, DiVinE model checker will be able to compute the fastest trace to the goal states which constitutes a time optimal schedule, *Path 3 with cost = 8*: $S_0 \xrightarrow{t[0]} S_1 \xrightarrow{t[1]} S_2 \xrightarrow{t[2]} S_3 \xrightarrow{t[3]} S_4$.The advantages of encoding control guards and execution of model with DiVinE are: find the (near) optimal results fast without exploring the full state-space; DiVinE can verify much larger system models and finish the verification in significantly less time in comparison with the well-known sequential model checkers.

C. Plant Specifications Modelling

The modelling and analysis of piece-wise manufacturing (PWM) systems with varying topologies and resource/performance constraints can be supported by applying the UTA template based modelling approach. We propose constructing the model from two types of items, (i) from processes that are instances of a generic machine template 3 and from (ii) passive resources/intermediate products which are needed to initiate the process or which are products of a process. The machine template is parametrized w.r.t. process flow topology, energy constraints (function powerConsumM()), machine performance characteristics (OpDur) and the amounts of workpieces consumed and produced by each machine during its processing cycle. Similarly, maintenance constraints are introduced by instantiating the parameters such as maximum period between required maintenance sessions (MtPeriod), the duration of maintenance (MtDur) and human resource needed for the maintenance. Due to the need for flexible modelling of processes all time related parameters are specified as intervals with their explicit lower and upper bounds. Time intervals for processing cycle, maintenance duration and period are represented using arrays indexed by machines to make the model adjustable for various process configurations. The parameter M denotes the number of machines in the process. The instances of Machine template are processes such as Making Ingot & Preheating, Transfering Ingots, Milling, Rolling & Finishing. The workpieces transferred between these processes are Ingots, hot aluminium stripes, cooled stripes and sheets.

A machine can be in one of the following states: idle, performing a processing cycle, being under the maintenance, or out of order due to the exceeded maximum allowed maintenance deadline. A processing cycle can be started when there is available minimum amount of resources (specified in the array `in`) needed for processing, whereas workpieces can be of different type and of different numbers (the elements of array `in` are indexed by machine numbers

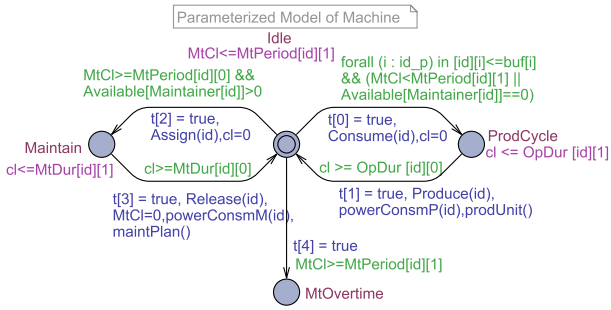


Fig. 3: Parameterized Model of manufacturing systems with maintenance constraints

and types of work pieces). Similarly, the array `out` specifies the workpieces produced as a result of processing cycle.

To model the processed items between processing cycles the numbers of items are represented as their quantities in virtual buffers (array `buf`) and the buffers are indexed by the types of workpieces. Second condition for starting a machining cycle is the constraint that maximum allowed period between maintenance activities has not been exceeded. Third constraint does not allow starting the processing by machine when the maintenance session is going on.

As stated above, the result of a machining cycle is a set of new items that are either final products or consumables for other machines. The triggering conditions of the machine maintenance session are the availability of a relevant maintenance person and the time passed after previous maintenance is within interval [`MinPeriod`, `MaxPeriod`]. The array `Maintainer` stores the reference to the required profile of a maintenance person for a Machine. The array `Available` stores the numbers of service persons of different profile currently available. While starting the maintenance of a machine the corresponding number of available maintenance persons is decreased in the array `Available`, and increased respectively after the maintenance session is completed.

V. Analysis and Maintenance planning by model checking

Given a manufacturing system model built by using the instances of the Machine template one can analyze various performance and safety properties using parametric model checking [25].

A. Property 1

For instance, the query for estimating the feasibility of full production plan can be stated as LTL formula: where `buf[P-1]`

```
#define Goal1 (buf[P-1] >= ProdPlan)
#define Goal2 (forall(i : int[0,M-1])
      Machine(i).t[3] == true
#define Time (cl<=MaxDur)
#property G!(Goal1 && Goal2 && Time)
```

denotes the amount of end-products manufactured and `ProdPlan` denotes the amount of end-products expected by the plan. Also, plan the maintenance schedule of each machine atleast once during production schedule. The counterexample trace generated by DiVinE model checking tool provides the time-wise optimal schedule of completing the `ProdPlan` and the maintenance schedule during the manufacturing process. When the analysis addresses the limited store problems similar reachability queries can be specified and model checked with respect to the upper bound of store volume instead of constant `ProdPlan`.

Optimal Schedule Plan:

$$\texttt{Machine(0).Idle} \xrightarrow{t[0]=1} \texttt{Machine(0).prodCycle}$$
$$\xrightarrow{t[1]=1} \texttt{Machine(0).Idle} \xrightarrow{t[0]=1} \texttt{Machine(0).prodCycle}$$
$$\rightarrow \texttt{Machine(3).Idle} \xrightarrow{t[0]=1} \texttt{Machine(3).prodCycle}$$
$$\rightarrow \texttt{Machine(0).prodCycle} \xrightarrow{t[1]=1} \texttt{Machine(0).Idle}$$
$$\xrightarrow{t[0]=1} \texttt{Machine(0).prodCycle} \rightarrow \texttt{Machine(3).prodCycle}$$
$$\xrightarrow{t[1]=1} \texttt{Machine(3).Idle} \xrightarrow{t[0]=1} \texttt{Machine(3).prodCycle}$$
$$\texttt{Machine(0).prodCycle} \xrightarrow{t[1]=1} \texttt{Machine(0).Idle}$$
$$\texttt{Machine(1).Idle} \xrightarrow{t[2]=1} \texttt{Machine(1).Maintain}$$
$$\rightarrow \texttt{Machine(3).prodCycle} \xrightarrow{t[1]=1} \texttt{Machine(3).Idle}$$
$$\rightarrow \texttt{Machine(2).Idle} \xrightarrow{t[2]=1} \texttt{Machine(2).Maintain}$$
$$\rightarrow \texttt{Machine(1).Maintain} \xrightarrow{t[3]=1} \texttt{Machine(1).Idle}$$
$$\rightarrow \texttt{Machine(3).Idle} \xrightarrow{t[2]=1} \texttt{Machine(3).Maintain}$$
$$\rightarrow \texttt{Machine(0).Idle} \xrightarrow{t[2]=1} \texttt{Machine(0).Maintain}$$
$$\rightarrow \texttt{Machine(2).Maintain} \xrightarrow{t[3]=1} \texttt{Machine(2).Idle}$$
$$\rightarrow \texttt{Machine(3).Maintain} \xrightarrow{t[3]=1} \texttt{Machine(3).Idle}$$
$$\xrightarrow{t[0]=1} \texttt{Machine(3).prodCycle} \rightarrow \texttt{Machine(0).Maintain}$$
$$\xrightarrow{t[3]=1} \texttt{Machine(0).Idle Machine(3).prodCycle}$$
$$\xrightarrow{t[1]=1} \texttt{Machine(3).Idle}$$

B. Property 2

The production constraints can be stated as LTL formulas, where $ProdDemand$ and $UnitReserve$ denotes the demand and reserve of the number of processed items produced according to particular production plan during the time interval `MaxDur`. Considering $MaxprodUnits$ to be the number of pieces produced, the production and demand constraints are given by query:

```
#define Goal (MaxprodUnits >= UnitReserve + ProdDemand)
#define Time (cl<=MaxDur)
#property G!(Goal && Time)
```

Here, the counterexample trace generated provides the optimal schedule and allows production plan to supply the demand by taking care of reserved pieces stored in the buffer.

Optimal Schedule Plan:

Machine(0).Idle $\xrightarrow{t[0]=1}$ Machine(0).prodCycle
$\rightarrow$ Machine(1).Idle $\xrightarrow{t[0]=1}$ Machine(1).prodCycle
$\rightarrow$ Machine(0).prodCycle $\xrightarrow{t[0]=1}$ Machine(0).Idle
$\rightarrow$ Machine(0).Idle $\xrightarrow{t[0]=1}$ Machine(0).prodCycle
$\rightarrow$ Machine(1).prodCycle $\xrightarrow{t[0]=1}$ Machine(1).Idle
$\rightarrow$ Machine(1).Idle $\xrightarrow{t[0]=1}$ Machine(1).prodCycle
$\rightarrow$ Machine(3).Idle $\xrightarrow{t[0]=1}$ Machine(3).prodCycle
$\rightarrow$ Machine(0).prodCycle $\xrightarrow{t[0]=1}$ Machine(0).Idle
$\rightarrow$ Machine(1).prodCycle $\xrightarrow{t[0]=1}$ Machine(1).Idle
$\rightarrow$ Machine(3).prodCycle $\xrightarrow{t[0]=1}$ Machine(3).Idle

C. Property 3

Similarly, the power balance constraints can be specified as LTL formula:

```
#define Goal1 (forall(i : int[0,M-1])
      powerConsumed_Global[i]) <= Power_G + Power_R
#define Goal2 (MaxprodUnits >= UnitReserve + ProdDemand)
#define Goal3 (forall(i : int[0,M-1])
      Machine(i).t[3] == true
#define Time (cl<=MaxDur)
#property G!(Goal1 && Goal2 && Goal3 &&Time)
```

where `Power_G`, `Power_R` and `powerConsumed_Global` denotes the power generated or bought, renewable generation in the production plan, and demand (total power consumption) at time `Cl`, respectively. Here, the fastest trace generated provides the optimal power consumption for particular production plan during the time interval `[0, MaxDur]`.

CONCLUSION

This investigation proposed a method to schedule maintenance operations using formal methods considering power balance and production constraints in process industries. We have shown that the production process optimization can be stated as a nonlinear optimization problem involving binary and real variables. The formulation includes maintenance schedules along with power balance constraints. Second, a technique using formal methods was proposed to check the feasibility of the scheduling problem, using guided model checking (construction of control guards by offline statistical analysis based on the given model and the test purposes) algorithm with DIVINE model checker. Combining the solution obtained from formal methods as initial feasible solution to the optimization models is the future course of the investigation.

ACKNOWLEDGMENT

This research is partially supported by Project IUT33-13 “Strong warranties software methodologies, tools and processes”and doctoral studies TTU, Estonia.

REFERENCES

[1] Srinivasan, S., Grobmann, D., Del Vecchio, C., Balas, V. E., & Glielmo, L. (2015, December). *Enabling technologies for Enterprise Wide Optimization. In Industrial and Information Systems (ICIIS)*, 2015 IEEE 10th International Conference on (pp. 434-439). IEEE.

[2] Y. Abdedda, E. Asarin, and O. Maler. *Scheduling With Timed Automata*. Theoretical Computer Science, 354(2):272 300, 2006.

[3] G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, and J.M.T. Romijn. *Efficient Guiding Towards Cost-Optimality in UPPAAL*. In Proceedings of TACAS 2001, volume 2031 of LNCS, pages 174188. Springer, 2001.

[4] G. Behrmann, K.G. Larsen, and J.I. Rasmussen. *Optimal Scheduling Using Priced Timed Automata*. SIGMETRICS Performance Evaluation Review, 32(4):3440, 2005.

[5] K.G. Larsen. *Resource-Efficient Scheduling for Real Time Systems*. In Proceedings of EMSOFT 2003, volume 2855 of LNCS, pages 1619, 2003.

[6] P. Niebert, S. Tripakis, and S. Yovine. *Minimum-time reachability for timed automata*. In Proceedings of MED 2000. IEEE Computer Society Press, 2000.

[7] K.G.Larsen,P. Pettersson ,and W. Yi. *UPPAAL in a Nut-shell*. Springer International Journal of Software Tools for Technology Transfer, 1(1+2):134-152,1997

[8] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004

[9] O. Maler. *Timed automata as an underlying model for planning and scheduling*. In M. Fox and A. M. Coddington, editors, AIPS Workshop on Planning for Temporal Domains, pages 6770, 2002.

[10] A. Maffei; S. Srinivasan; P. Castillejo, J. F. Martinez, L. Iannelli, E. Bjerkan and L. Glielmo, *A Semantic Middleware Supported Receding Horizon Optimal Power Flow in Energy Grids*, in *IEEE Transactions on Industrial Informatics*, vol. no.99, 2017.

[11] Verrilli, Francesca, et al., *Model Predictive Control-Based Optimal Operations of District Heating System With Thermal Energy Storage and Flexible Loads.* IEEE Transactions on Automation Science and Engineering, vol. 14, no.2, 2017, pp.547-557.

[12] Lixia Ji *Steel Production Scheduling Based on Priced Timed Automata*, 2352-5401, March 2015.

[13] D. Pal and J. Vain,*Generating optimal test cases for real-time systems using DIVINE model checker,* 2016 15th Biennial Baltic Electronics Conference (BEC), Tallinn, 2016, pp. 99-102. doi: 10.1109/BEC.2016.7743738

[14] Srinivasan, S., Buonopane, F., Ramaswamy, S., & Vain, J. (2014, November). *Verifying response times in networked automation systems using jitter bounds*. In Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on (pp. 47-50). IEEE.

[15] Srinivasan, S., Buonopane, F., Vain, J., & Ramaswamy, S. (2015). *Model checking response times in Networked Automation Systems using jitter bounds*. Computers in Industry, 74, 186-200.

[16] Balasubramaniyan, S., Srinivasan, S., Buonopane, F., Subathra, B., Vain, J., & Ramaswamy, S. (2016). *Design and verification of Cyber-Physical Systems using TrueTime, evolutionary optimization and UPPAAL*. Microprocessors and Microsystems, 42, 37-48.

[17] R. Alur and D. Dill. *A theory of timed automata*. Theoretical Computer Science B, 126:183235, 1994.

[18] ZHOU Qing-lei,JI Li-xia. *Model checking of real-time systems based on UPPAAL*. Computer Applications, 2004.

[19] Behrmann, G.; Fehnker, A.; Hune, T.; Larsen, K.; Pettersson, P.; Romijn, J.; and Vaandrager, F. 2001a. *Minimum cost reachability for priced timed automata*. Lecture Notes in Computer Science 2034 :pp. 147.

[20] Alur, R.; La Torre, S.; and Pappas, G. 2001. *Optimal paths in weighted timed automata*. Lecture Notes in Computer Science 2034:pp. 4962.

[21] Moshe Y. Vardi & Pierre Wolper (1986): *An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)*. In: LICS. IEEE Computer Society, pp. 332344.

[22] J. Barnat, L. Brim and V. Havel, *DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs,* in Computer Aided Verification (CAV 2013), vol. 8044, Springer, 2013, pp. 863-868.

[23] Barnat, J., Brim, L., eka, M., and Rokai, P. *DIVINE: Parallel Distributed Model Checker (Tool paper)*. In Parallel and Distributed Methods in Verification and High Performance Computational System. Biology (HiBi/PDMC 2010), pages 47. IEEE, 2010.

[24] M. A. Basit-Ur-Rahim, F. Arif and J. Ahmad, *Modeling of real-time embedded systems using SysML and its verification using UPPAAL and DiVinE*, 2014 IEEE 5th International Conference on Software Engineering and Service Science, Beijing, 2014, pp. 132-136. doi: 10.1109/ICSESS.2014.6933529

[25] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. isbn: 026202649X, 9780262026499.

Appendix 5

V

Pal, D.; Vain, J. (2016). Generating optimal test cases for real-time systems using DIVINE model checker. BEC 2016 : 15th Biennial Baltic Electronics Conference, Tallinn University of Technology, October 3-5, 2016 Tallinn, Estonia, 99–102

Generating Optimal Test Cases for Real-Time Systems using DIVINE Model Checker

Deepak Pal,
Elvior LLC
Software Test Automation
Tallinn, Estonia
deepak.pal@elvior.ee

Jüri Vain
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia
juri.vain@ttu.ee

***Abstract*—The automatic generation of witness and counterexample is considered as the key advantage of model checking. It provides a useful source of diagnostic information and a basis for automated test generation. However, some of the witness traces may be unreasonably long and highly redundant that makes achieving test purpose even for small systems inefficient. This paper presents a technique for automated generation of optimal test cases from the specification model. The proposed technique performs reachability analysis using DIVINE (a distributed-memory model checker) to generate optimal test cases. It is implemented in the form of control guards to guide model on-the-fly towards test goals which are constructed offline by statistical analysis based on model and test purpose. To prove correctness and performance of the technique, we demonstrate a case study on web application which performs several types of tests and we compare results against UPPAAL.**

I. INTRODUCTION

Extensive research has been done in the field of formal verification of real-time systems to solve realistic scheduling problems. The common idea of these works is to reformulate a scheduling problem to a reachability problem that can be solved by verification tools. Model checking is the well-known representative of formal verification techniques. The principle of witness and counterexample is considered as the key advantage of model checking. It provides a useful source of diagnostic information and a basis for automated test generation. A principle problem with current model checking tools is that a very large (generally infinite) number of test cases can be generated from even small models. It is easy to imagine small systems having enormous state spaces which makes model checking a very expensive method in both time and memory usage. Every additional variable can potentially grow the state space of the system - this is called state explosion problem.

From an industrial perspective, model checking is a promising tool not only for formal verification but also for automated generation of test cases. It is helpful to improve the quality and effectiveness of testing, and to reduce its cost. The current state of practice is not only restricted to test automation but focuses on automatically generating quality tests from models. Academic tools like HyTech [3], Kronos [2], UPPAAL [1], Spin [13] etc. offer different types of verification facilities (on-line verification, symbolic verification, abstraction techniques, etc.) which supports only small applications which may restrict their use on industrial scale.

In [4], authors introduce an open-source distributed-memory model checking tool-DIVINE [11] that propose a way to cope with large state spaces by using more computation power. Although there are many widespread tools for model checking but none of them can properly use parallelization for better performance [7]. Several experimental parallel model checkers were developed, but they are not as popular as the conventional sequential tools. It is often caused by the way of implementation and software distribution. For instance the distributed version of UPPAAL is not publicly available neither as open source application nor as ready-to-install package. It is very hard to extend because it is not modular, i.e. a single change on one place of the source code can imply many modifications on other places [7]. Many other tools exist only as experimental software that are even not well documented. To fill this gap the distributed model checking tool DIVINE employs an aggregate power to verify large systems models with better efficiency and memory usage. Its verification power with distributed algorithms is beyond the capabilities of sequential tools [11]. In this paper we present a technique for automatically generating optimal test cases for real-time systems using DIVINE model checker as reachability analysis tool. The main contributions of the paper are:

- Demonstration of the use of diagnostic traces generated by DIVINE as a basis for automated test generation.
- Construction of control guards in the model for generating optimal traces. This is due to fact that DIVINE provides only standard reachability, it always returns the first trace it finds, it does not optimize the traces in any way.
- Experimental results prove that the proposed approach allows scaling up the scope of testing for models with large state space.

This paper is organized as follows. Section 2 formally defines modelling formalism, model checking with DIVINE and LTL properties, Section 3 explains the implementation of constructing the control guards and encoding them on transitions, Section 4 demonstrates the experimental results

978-1-5090-1393-7/16/$31.00 ©2016 IEEE

and its comparison with UPPAAL.

II. PRELIMINARIES

A. Modelling System and Verification with DIVINE

DIVINE is an explicit-state linear temporal logic (LTL) automata-based verification tool for reachability analysis of discrete distributed systems [10]. It employs an aggregate power to verify large systems models with better efficiency and memory usage [11], [12]. To run DIVINE and verify model properties, a LTL file is provided alongside the model xml file. LTL formula are loaded from this file and when a specific property is chosen, it is negated and a Büchi automaton is created. The Büchi automaton is multiplied with the timed automata on-the-fly to create a transition system on which the reachability analysis or accepting cycle detection algorithms are run [4], [5]. DIVINE runs the reachability analysis to find an error state or a time deadlocks. It performs universal verification only, i.e. it decides whether all runs meet given conditions. However, negation of existential formula is a universal formula. Therefore, the CTL specification in the form $A[]p$ or $A <> p$ can be directly translated into equivalent LTL formula Gp or Fp respectively. $E <> p$ can be translated to $G!p$ and $E[]p$ to $F!p$, but due to the mapping to universally quantified formuli the validation turns to falsification, i.e. the formulas in CTL and corresponding formula in LTL are not equivalent, in fact, they have exactly opposite meaning; hence when DIVINE says a formula $F!p$ does not hold in some model, UPPAAL says that $E[]p$ is satisfied and vice versa. In the case of existential CTL formula that holds, DIVINE reports a counterexample to corresponding LTL formula, which is actually a witness for the original CTL formula [11].

B. Modelling

We demonstrate how a network of timed automata can be used for modelling the System Under Test (SUT). Since DIVINE does not support modelling formalism for real-time systems, we preferred UPPAAL Timed Automata (UTA)[1] which have become the standard modelling language for real-time systems. Therefore we use UTA as the modelling language and DIVINE model checker as the reachability analysis tool. We model the SUT behavior using UTA patterns called

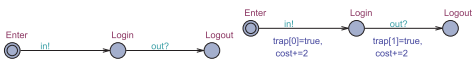


Fig. 1. Example of extended model

actions. We extend the model with set of boolean variables called traps as defined in [6] and cost associated on each transitions. The traps are encoded on transitions of model as boolean update functions with initial value set to false. The traps are executed when the transition is triggered during the execution and set to true. A set of traps can be used to see the path taken by test run. The cost of taking an action transition is the price associated with the transition and the cost of a trace is simply the accumulated sum of costs of its action transitions. The objective is to determine the minimum cost of traces ending in a goal state. An example of extended model with trap and cost variables is shown in Figure 1. We modelled the real SUT *Registration form* shown in Fig 2,

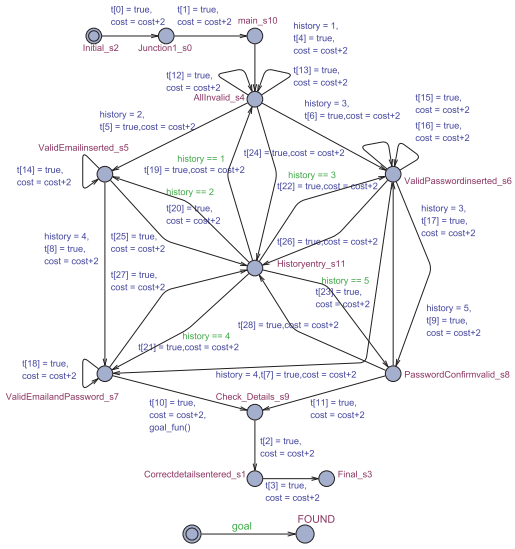


Fig. 2. SUT Model of Registration Form and Goal Model

which comprises of three fields i.e. enter username, password and confirm password. The correct details enable the register button. To test functionality of such system we need to perform some positive and negative tests. Our goal is not to perform all possible test cases which can be expensive as tester wants to check only specific functionality. To get best optimal and fastest test possible that checks specific functionality is the first priority.

C. From Diagnostic Traces to Test Cases

A most common approach to the test generation is to formulate an informal set of test purposes transformed into some property of system such that model can be used to generate test cases for each property. A test purpose is specific property of system that tester wants to observe on the system under test.

The test purpose can be directly formulated as a simple state reachability of *property* (Goal.Found) also known as single purpose test case generation. In Figure 2, a test purpose is assigned with *goal_fun* and the problem is to find an optimal path that reaches the goal state. An example of test purpose (LTL property) of model is expressed as:
♯define objective (Goal.Found) ♯property G!(objective).
Here ♯define gives a symbolic name for an atomic proposition, and ♯property specifies a single LTL formula.

- *Test-Purpose1*: Verify that the after entering the valid email, password enter in *confirm password* field match to correct password.

- *Test-Purpose2*: Verify that user can register successfully.

Test purpose 1 and 2 can be formulated as reachability property with LTL formula $G!p$ (formula described in subsection A) i.e. *G! (Goal.Found)* which means eventually the registration form automaton enters the state *valid email and password* and thereafter it eventually enters the state *correct details entered*.

Generally, on large scale systems testers are interested in generating a test suites that guarantee that the SUT model is tested thoroughly and covered in a certain way. To achieve certain level of quality and thoroughness in test generation process the model should be covered by test runs so that possibly many test purposes (*goal_fun*) that can label on transitions/locations are visited. Such test cases or test suites can be generated automatically from natural coverage criteria such as *edge* or *transition* coverage of timed automata model for more reading refer to [9]. We feed the SUT and GOAL (test purpose) models to DIVINE to verify the property of SUT using command line options. When computing the traces that satisfy reachability given properties, DIVINE provides only standard reachability i.e. it always returns the first trace it finds. This might not be the shortest/fastest test case since it does not optimize the traces in any way. Also, it does not terminate upon finding the path to the reachability problem and continues until there are no more states, alternatively model checker ends in deadlock which is common issue with other model checking tools (UPPAAL) as well. Consequently, DIVINE performs the random exploration of the state space, which generates the test cases that may be unreasonably long and may leave the test purpose unachieved. We have run the query several times for SUT model shown in Fig 2 using DIVINE command line options and to reach the goal by traversing the different transitions and generated traces are as follows:

- *Traces generated 1*: *Goal = Found*, cost = 24;
  Initial_s2 $\xrightarrow{t[0]}$ Junction1_s0 $\xrightarrow{t[1]}$ main_s10 $\xrightarrow{t[4]}$ AllInvalid_s4 $\xrightarrow{t[13]}$ AllInvalid_s4 $\xrightarrow{t[5]}$ ValidEmailinserted_s5 $\xrightarrow{t[25]}$ Historyentry_s11 $\xrightarrow{t[20]}$ ValidEmailinserted_s5 $\xrightarrow{t[8]}$ ValidEmailandPassword_s7 $\xrightarrow{t[18]}$ ValidEmailandPassword_s7 $\xrightarrow{t[10]}$ CheckDetails_s9 $\xrightarrow{t[2]}$ Correctdetailsentered_s1 $\xrightarrow{t[3]}$ Final_s3 done.

- *Traces generated 2*: *Goal = Found*, cost = 24;
  Initial_s2 $\xrightarrow{t[0]}$ Junction1_s0 $\xrightarrow{t[1]}$ main_s10 $\xrightarrow{t[4]}$ AllInvalid_s4 $\xrightarrow{t[13]}$ AllInvalid_s4 $\xrightarrow{t[5]}$ ValidEmailinserted_s5 $\xrightarrow{t[8]}$ ValidEmailandPassword_s7 $\xrightarrow{t[18]}$ ValidEmailandPassword_s7 $\xrightarrow{t[27]}$ Historyentry_s11 $\xrightarrow{t[21]}$ ValidEmailandPassword_s7 $\xrightarrow{t[10]}$ CheckDetails_s9 $\xrightarrow{t[2]}$ Correctdetailsentered_s1 $\xrightarrow{t[3]}$ Final_s3 done.

The traces generated show that model checker chooses the longer path to reach the goal with cost = 24 instead of shortest/fastest path: Initial_s2 $\xrightarrow{t[0]}$ Junction1_s0 $\xrightarrow{t[1]}$ main_s10 $\xrightarrow{t[4]}$ AllInvalid_s4 $\xrightarrow{t[5]}$ ValidEmailinserted_s5 $\xrightarrow{t[8]}$ ValidEmailandPassword_s7 $\xrightarrow{t[10]}$ CheckDetails_s9; which costs = 12 is optimal to reach the goal. Another issue we found is that DIVINE does not stop immediately after finding goal, it continues until a deadlock.

To overcome the problem of long test cases which are not optimal, we followed the reactive planning tester (RPT) algorithm discussed in [6] and extend it to generate sub-optimal (w.r.t. given planning horizon) test cases which find the minimum cost of reaching a goal state and stop the search as soon as it reaches to goal state.

III. ALGORITHM FOR GENERATING CONTROL GUARDS

A technique of finding optimal traces for test is implemented in the form of control guards which are generated based on RPT algorithm. A control guard of a transition of the model is constructed to meet the following requirements:

- each transition to be enabled by its control guard must guide the test run to locally optimal w.r.t achieving the test purpose from the current state of the model and
- the model should terminate immediately after the test purpose is achieved (Goal.Found).

The control guards for each transition are constructed by offline static analysis based on the given model and the test purposes. The generated control guards are conjoined with original guard conditions on transitions of the model and feed to DIVINE where controls are evaluated in every state when the selection between alternative outgoing transitions should be made. The execution of the model with control guards finds the fastest and efficient path to goal state and leads to a witness trace with optimal cost.

IV. EXPERIMENT & RESULTS

In the previous section we discussed the basic idea used for constructing control guards to compute optimal test cases. In this section we present experiments performed on SUT model. We feed the models to both UPPAAL Cora [14] and DIVINE and compare results obtained from these model checkers. It is well known that UPPAAL Cora provides optimized results if we ignore state explosion problem on large scale. However our aim is to show better performance and efficient memory usage with DIVINE for large scale models used in industries. We extend the SUT model shown in Figure 2 with control guards on each transition and feed to DIVINE. The generated traces are: Initial_s2 $\xrightarrow{t[0]}$ Junction1_s0 $\xrightarrow{t[1]}$ main_s10 $\xrightarrow{t[4]}$ AllInvalid_s4 $\xrightarrow{t[5]}$ ValidEmailinserted_s5 $\xrightarrow{t[8]}$ ValidEmailandPassword_s7 $\xrightarrow{t[10]}$ CheckDetails_s9; which is optimal with minimum cost = 12 and DIVINE stops further execution immediately reached to goal state. In comparison to above results, UPPAAL Cora performs the similar actions and generates the fastest traces with same minimum cost = 12. To compute the performance of our implementation, we have run the several tests on different models and properties. Variants of registrationForm.xml, lightSwitch.xml (Light Switch On/Off model) and telema.xml (Telema model) are implemented and the properties being verified on them are G!(GOAL.FOUND). Here, registration form model is same as shown in Fig 2. Light

TABLE I
EXPERIMENT RESULTS WITH DIVINE AND UPPAAL

Model	DIVINE Time [s]	DIVINE Memory [KB]	DIVINE State count	UPPAAL Time [s]	UPPAAL Memory [KB]
$regForm1^p$	0.042	36576	14	0.025	32168
$regForm2^p$	0.062	36748	18	0.004	32176
$regForm3^p$	0.05	36740	12	0.003	33584
$lightSwitch1^p$	108	36524	8021	73.97	2033620
$lightSwitch2^p$	840	45024	64021	*	*
$telema1^p$	0.70	124612	10	*	*
$telema2^p$	72	131132	75	*	*

p —LTL property expressed as $G!(\phi)$
* —Out of memory.
See http://bugsy.grid.aau.dk/bugzilla3/show_bug.cgi?id=63

Switch model is simple model with 3 states and 7 transitions consisting of 4 loops. The idea is to switch On and Off light 1000 times or more to check its durability and generate tests for the same. Telema model is huge model with 790 states and 1432 transitions which model the behavior of real system (http://telema.ee/en) where tests cases/suites are generated for different test purposes. These models are also available in [8].

All tests were performed on machine with Intel(R) core(tm) i5-4200m cpu@2.50ghz using DIVINE 3.3.2 release and UPPAAL 4.1.19. Table 1 shows the time and memory required to perform verification using DIVINE and UPPAAL and number of states visited by DIVINE.

The table 1 summarizes the results. The results for small models turned out to be as expected —DIVINE consumes more memory but requires comparable amount of time. This happens due to DIVINE executable alone needs around 140MB of memory to run and lack of memory optimizations because DIVINE runs with default setting (OWCTY or reachability algorithm) whereas UPPAAL runs on memory optimizations. Also above models use meta variables, which are considered as regular variables by DIVINE. It means that DIVINE might have generated larger number of states than UPPAAL. On larger models or models with loops/cycles, the DIVINE shows quite varied results. When run on light switch model with test purpose to switch On/Off light for 1K times DIVINE need less memory than UPPAAL but is slower than UPPAAL. On the other hand, results for lightSwitch2.xml (On/Off 8K times) show that DIVINE performs well on larger variants where UPPAAL shows out of memory. Similarly for Telema model UPPAAL shows out of memory because of large state space where DIVINE shows satisfactory results.

V. CONCLUSION

In this paper, we have presented a technique based on RPT algorithm originally introduced in [6] for generating optimal test cases using DIVINE model checker. We demonstrated that counterexample generated by DIVINE can be a useful source of diagnostic information and a basis for automated test generation. However, DIVINE provides only standard reachability that always returns the first trace it finds, and does not optimize the traces in any way. We presented a technique that performs reachability analysis using DIVINE to generate optimal test cases from extended specification model with encoded control guards on transitions. To show the correctness and performance of the technique, we demonstrated web application case study on which we performed several types of tests and compared the results against UPPAAL Cora. We outlined the limitations of model checking tools which can generate test cases only for small sized systems (due to state explosion problem) and presented the DIVINE as powerful distributed parallel model checker whose verification power with distributed algorithms is beyond the capabilities of sequential tools. However we found several ways to improve our technique and demonstrated its extendability for testing large scale real-time systems.

ACKNOWLEDGMENT

This research has been supported by European Union Regional Development Fund.

REFERENCES

[1] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. Int. Journal on Software Tools for Technology Transfer, 1(12):134152, October 1997.
[2] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In Proc. of the 10th Int. Conf. on Computer Aided Verification, number 1427 in Lecture Notes in Computer Science, pages 546550. SpringerVerlag, 1998.
[3] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybird Systems. In Orna Grumberg, editor, Proc. of the 9th Int. Conf. on Computer Aided Verification, number 1254 in Lecture Notes in Computer Science, pages 460463. SpringerVerlag, 1997.
[4] Barnat, J., Brim, L., eka, M., and Rokai, P. DIVINE: Parallel Distributed Model Checker (Tool paper). In Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010), pages 47. IEEE, 2010.
[5] Barnat, J., Brim, L., Havel, V., Havlek, J., Kriho, J., Leno, M., Rokai, P., till, V., and Weiser, J. DIVINE 3.0 An Explicit-State Model Checker for Multithreaded C & C++ Programs. In Computer Aided Verification (CAV 2013), page 6. LNCS, 2013. To appear.
[6] Vain, J.: Raiend, K.; Kull, A.; Ernits, J. (2007). Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In: ASE'07 : 2007 ACM/IEEE International Conference on Automated Software Engineering, Atlanta, Georgia, November 5-9, 2007, proceedings: 22nd IEEE/ACM International Conference on Automated Software Engineering. ACM Press, 363372.
[7] Pavel Šimeček, DIVINE- Distributed Verification Environment, Master Thesis. http://is.muni.cz/th/51636/fi_m/master_thesis.pdf
[8] Personal WebPage Deepak Pal http://research-deepak.com/distributed_testing.html
[9] Anders Hessel, Kim Guldstrand Larsen, Brian Nielsen, Paul Pettersson, Arne Skou: Time-Optimal Test Cases for Real-Time Systems. FORMATS 2003: 234-245.
[10] Moshe Y. Vardi & Pierre Wolper (1986): An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In: LICS. IEEE Computer Society, pp. 332344.
[11] DIVINE Model Checking for Everyone https://divine.fi.muni.cz/
[12] J. Barnat and L. Brim and I. Cerna and P. Moravec and P. Rockai and P. Simecek: DiVinE – A Tool for Distributed Verification (Tool Paper) Computer Aided Verification, Springer Berlin / Heidelberg, 2006, volume 4144/2006 of LNCS, 278-281. [http://www.fi.muni.cz/ xbarnat/publications/cav2006-.pdf]
[13] The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, ISBN 0-321-22862-6, 608 pgs, 2004.
[14] Behrmann, Gerd ; Larsen, Kim Guldstrand ; Rasmussen, Jacob Illum: "Optimal scheduling using priced timed automata", in SIGMETRICS Performance Evaluation Review.

Appendix 6

VI

Muthukumar, N.; Srinivasan, Seshadhri; Ramkumar, K.; Pal, Deepak; Vain, Jüri; Ramaswamy, Srini (2019). A model-based approach for design and verification of Industrial Internet of Things. Future Generation Computer Systems, 354–363

Future Generation Computer Systems 95 (2019) 354–363

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A model-based approach for design and verification of Industrial Internet of Things

Muthukumar N. [a], Seshadhri Srinivasan [b,*], K. Ramkumar [a], Deepak Pal [c], Juri Vain [c], Srini Ramaswamy [d]

[a] Electric Vehicle Engineering and Robotics (EVER) Lab, SASTRA Deemed to be University, Thanjavur, 613401, India
[b] Berkeley Education Alliance for Research in Singapore, Singapore 138602, Singapore
[c] Department of Computer Science, Tallinn University of Technology, Tallinn, Estonia
[d] ABB Inc., USA

HIGHLIGHTS

- An IIoT Architecture for process industry is proposed.
- Model-Based approach for design and verification of IIoT is proposed.
- Multiple-view modelling approach is used in model-based design and verification.
- Work-flow for model-based design is presented.
- Demonstrates the MBE approach in processing industry.

ARTICLE INFO

Article history:
Received 5 September 2018
Accepted 7 December 2018
Available online 6 January 2019

Keywords:
Industrial Internet of Things (IIoT)
Model-Based Engineering (MBE)
Verification
Process industries
IoT architecture

ABSTRACT

This investigation presents an Industrial Internet of Things (IIoT) architecture and a Model-Based Engineering (MBE) approach for design, verification, and auto-code generation of control applications in process industries. The IIoT architecture describes the hardware components, communication modules, and software. It emerges as a major enabler for providing open connectivity to process industry which provides greater data-aggregation, visibility, availability, flexible control, and cloud-connectivity. The MBE approach is based on multiple views of the systems with each domain model describing a particular view. The multi-view modelling approach is used to perform design and verification of the IIoT enabled control in process industries. We show that such an integration of MBE, cloud-computing, and IIoT provides certain desirable features such as plug-and-play control and on-the fly verification which are lacking in the process industry. The proposed MBE approach and IIoT architecture are illustrated on the quadruple tank process, a benchmark problem in control. Our deployment results verify the benefits envisaged by IIoT, cloud, and MBE integration.

1. Introduction

Combining the Internet of Things (IoT) with cloud-intrinsic capabilities can transform the way industrial automation systems are designed, deployed and managed currently in process industries [1–3]. While the cloud offers capabilities such as virtualization, scalability, lifecycle management, and multi-tenancy, the IoT compliments it using its open connectivity and emergent computing environments (e.g., fog computing). In addition, the cloud offers attractive delivery models such as software-as-a-service, platform-as-a-service, and infrastructure-as-a-service with different deployment models. Consequently, many desirable features such as increased flexibility, adaptability, data-visualization, enterprise-wide communication, intelligence, and agility can be realized on low-power electronic devices. Besides, the cloud can host a variety of auxiliary services that can enhance the automation capabilities and promote smart manufacturing. Many recent investigations have stressed the need for transforming the cloud–IoT integration to deployment (see, [4–6] and references therein). However, industrial automation lacks engineering approaches and tools to accomplish this integration. Moreover, their deployment possess strict challenges due to hardware limitations of the IoT components such as real-time performance, reliability, and safety. Arguably, the IoT-based devices cannot fully substitute the legacy automation

* Corresponding author.
E-mail addresses: muthukumar.n@sastra.ac.in (Muthukumar N.), seshadhri.srinivasan@bears-berkeley.sg (S. Srinivasan), ramkumar@eie.sastra.edu (K. Ramkumar), Deepak.pal@ttu.ee (D. Pal), juri.vain@ttu.ee (J. Vain), srini@ieee.org (S. Ramaswamy).

https://doi.org/10.1016/j.future.2018.12.012
0167-739X/

systems, but they can be deployed in tandem with them to perform specific/specialized tasks. This requires frameworks that consider both legacy and IoT devices in one framework.

Engineering industrial automation systems has been focus of many investigations and methods based on component-based [7], formal models [8], agent-based [9], service-oriented architecture (SoA) [10], design patterns [11] and Model-Based Engineering (MBE) [12] have been proposed. Vyatkin [13] provides a good review on these approaches. Notwithstanding these developments, the automation software complexity and the functionalities realized using them have grown steadily. This, the industries discern, will increase the design, validation and verification costs significantly. Moreover, design upgrades and post design validations are proving costlier. In this backdrop, the Model-Based Engineering (MBE), an approach using models to design software and perform component testing emerges as a promising solution. As they automate the design process through auto-code generation capabilities. Further, design validation can be performed early during the life-cycle. The use of MBE approach for code-generation in legacy industrial automation systems has been studied in [14]. Similarly, to handle the complexity of industrial automation system with entangled behaviours from various domains, artefacts, and interactions, multi-domain models have been studied in [15]. As for Industrial IoT, an UML (Unified Modelling Language) profile for IoT in manufacturing industry was presented in [16]. The use of semantic technologies adding meaning to machine-to-machine communication using ontologies of interlinked terms, concepts, relationships and entities was investigated in the context of IIoT in [17]. These investigations either model legacy systems or IoT systems without involving cloud features.

More recently, combining cloud-intrinsic features with IIoT for providing enterprise-wide connectivity has been studied in [18, 19]. The IMC-AESOP project [20] extended the engineering methods based on Object Oriented and Aspect Oriented approach to industrial automation [21] using formal modelling extensions. Similarly, the use of agent-based approaches for cloud integrated IoT systems was studied in [22]. However, the use of MBE approach for cloud-based IIoT starting from the model to the deployment is currently not available to our best knowledge.

This investigation addresses this research gap by proposing a multi-view model of industrial automation and an MBE approach for design and verification of cloud-based IIoT implementations in process industries. The main contributions of this investigation are: *(i)* An IIoT architecture that promotes cloud-based engineering of the process control applications, *(ii)* Multi-view models for industrial automation systems in process industries that include various participating domains, artefacts, and interactions, *(iii)* A MBE approach for designing and verifying cloud-based IIoT, *(iv)* a workflow for performing Model-Based Design (MBD) and verification in emergent IIoT paradigm to realize sophisticated controllers, e.g., model predictive controller [23], *(v)* Present the advantages of the proposed architecture to perform plug-and-play control, on-the-fly verification, and smart manufacturing, and *(vi)* Demonstrate the MBE approach on a quadruple tank process applications.

The paper is organized into six sections. Section 2, presents the IIoT architecture and the MBE approach is discussed in Section 3. The cloud-enabled flexibilities are discussed in Section 4. Section 5 presents the deployment results of the IIoT. Conclusions and future course of investigation are discussed in Section 6.

2. Proposed IIoT architecture

The architecture that enables MBE for cloud-based IIoT is shown in Fig. 1. It consists of three major blocks: plant-level automation, the IIoT gateway, and the automation cloud. The plant-level automation consists of conventional *Programmable Logic Controllers* (PLCs) and IoT based commercial-of-the-shelf (COTS) target platform. The PLC interfaces to the sensors using conventional industrial protocols (e.g., Modbus). While the COTS platform uses TCP based protocols such as Message Queue Telemetry Transport (MQTT) or Advanced Message Queuing Protocol (AMQP), wireless and other forms of dedicated communication (e.g., I2C) to interface the field devices. A gateway is used to communicate with COTS target platform and legacy protocols with an incompatible physical layer (e.g., Profibus PA). OPC UA is used for aggregating information from the conventional PLCs and field devices due to its prevalence in the automation industry. Further, its security and platform independence makes it a good choice for the IIoT.

The IIoT gateway has interfaces on one side to the plant-level automation, and on the other to the cloud. The IIoT core is the main component of the IIoT gateway that orchestrates different protocols, devices, applications and software routines. It collects data from OPC UA using a client and transfers to other devices using MQTT or AMQP extensions. The MQTT extensions, (i.e., services) are used to collect information from the MQTT broker (an entity that supplies information to all devices subscribing to it). The OPC UA client and MQTT extension perform both device and data management within the IIoT gateway. The FTP, web interfaces and web applications are used to communicate to plant-level devices and cloud. The IIoT gateway provides extensions for the cloud and hardware devices, data persistence (DP) for securing data delivery in events of communication failures, and a secured FTP for enhancing the application security. Here it should be clarified that the MQTT and AMQP are shown as communication links only for illustrative purposes of this investigation. The IIoT gateway can be used for other protocols as well with suitable modification.

The cloud-intrinsic features — DP, virtualization, communication interfaces, multi-tenancy, auxiliary application support and others are offered by the automation cloud. The cloud offers virtualization through model repositories and emulators. The model repositories consist of the processes and controller instances, topology, behavioural models of the devices and all other aspects required for performing MBE. Employing the communication interfaces, the cloud talks to the IIoT gateway through the MQTT and AMQP. To compliment the IIoT gateway, the cloud has FTP, HTTP and external interfaces for enabling file transfer, web applications and using third-party applications. The IIoT architecture simplifies the communication between legacy devices and IoT devices in the plant-floor and enables open connectivity between plant-floor and cloud. Therefore, the architecture promotes the implementation of the cloud-based IIoT.

3. Model driven engineering for Industrial Internet of Things

With the emergence of IIoT, the heterogeneity and networking capability of the hardware, and the proportion of system functionality realized using software has increased stupendously leading to an increase in the design space. Coupled with these developments, market influences requiring smart and flexible manufacturing are obligating a more flexible automation that provides upgrades/modifications with minimum engineering effort. As stated earlier, the MBE approach is more suitable in such scenarios as it raises the abstraction levels and automates the labour-intensive and error-prone tasks in the design, e.g., code-development [13]. This not only brings down the design cost, but enhances reusability, efficient data exchange, and verifiability of the system. Above all, the MBE promotes MBD and Model-Based Verification (MBV). Using these methods the design, validation and verification can be automated to a greater extent even from the cloud. However, the model of the industrial automation system by itself is complex due to the interaction of multiple domains and heterogeneous entities. There is a lack of tools, formalisms and semantics capable of incorporating semantic relations among the disciplines. Developing a

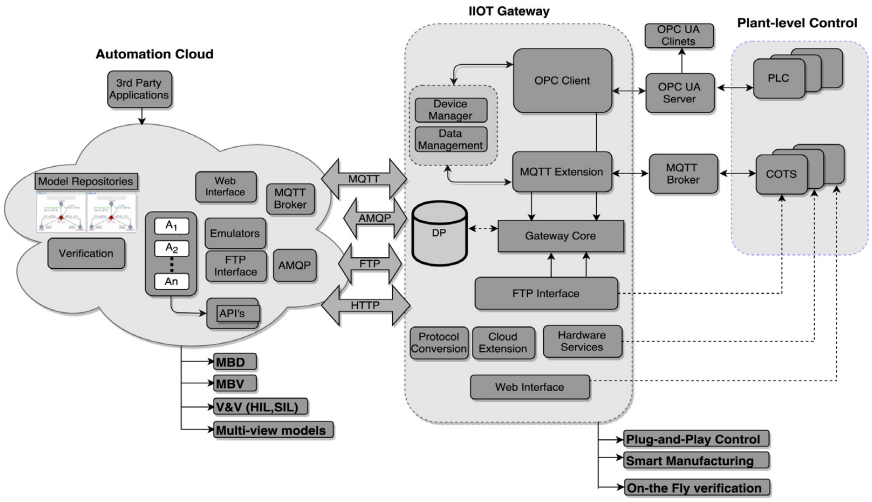


Fig. 1. Proposed IIoT architecture.

meta-model encapsulating all aspects of an industrial automation system is rather difficult. More recently, the use of multiple views for industrial automation systems have been investigated for industrial production units [24]. This investigation uses the multi-view modelling approach for performing MBE for cloud-based IIoT solutions.

3.1. Multi-View model for IIoT

Multiple views model is an emerging concept for building complex systems wherein different stakeholder's viewpoints are captured as domain models or *views* [25]. The multi-view model of cloud based IIoT has different but entangled views—devices, architecture, information, software, control, domains, behaviour and others. These different viewpoints need to be considered simultaneously for engineering IIoT systems. Consequently, system integration emerges as a key challenge due to potential contradictions or overlapping information among the views. Therefore model transformations and mapping are required for engineering systems with multiple views. This investigation uses a meta-modelling approach for capturing the different views.

The multi-view model of the industrial automation system and the different tools for obtaining these views are shown in Fig. 2. To integrate these different views, this investigation uses the AutomationML (Automation Markup Language)[1] (AML) for providing the topology view and uses it as a meta-model of the IIoT based automation systems [26]. The process industries with its various process stations are modelled in the AML using suitable abstractions. The AML provides XML/CAEX (Computer Aided Engineering Exchange) formats for the topology view and in addition provides the communication view through the *InterfaceLibraryClass*, wherein additional interfaces specific to IIoT are defined. The process views are obtained from the P and ID diagram, the controller design is modelled in Simulink using state-space/transition formalisms, the OPC UA provides the information models, the software design is modelled using UML and behavioural models based on statecharts. In addition to these models, there can be domain views that capture the formalisms and artefacts of the different domains (electrical, mechanical) modelled using suitable software tools, e.g., Dymola. The multi-view model forms the basis on which the MBD and MBV are performed.

[1] https://www.automationml.org/.

3.2. Workflow for model-based design and verification

The workflow for performing MBD and MBV from multi-view models is illustrated in Fig. 3. The P and ID's process view defined in the IEC 62424 standard is used as the starting point. It has three basic concepts: process control engineering requests (PCE-R), process control engineering function (PCE-f), and process control loop (optional). The PCE-R defines the requirements of the process control equipment. The PCE-R collects all information about the functional requirements. PCE-R and its unique ID are important specifications for the requirements diagram.

The AML model uses the PCE-R to create a meta-model of the entire process that can be later used to map different models. The ability to produce neutral XML/CAEX schema makes AML a suitable tool for information exchange between engineering applications. The *InstanceHierarchy* represents the entire automation project and it has the child nodes called the *InternalElements* that hold the attributes of the different properties of the object and have objects that hold the attributes need to describe them. Here, process control loop implies the unitary process description, e.g., level control of the tank. Each object in the *InternalElements* is associated with a *RoleClassLibrary* that provides the functional view of the object, an useful aspect for semantic classification. In addition, there is the *SystemUnitClassLibrary* defining the specific aspects of the process control application, e.g., height of the specific tank.

The communication interfaces are modelled using the AML basic *InterfaceLibrary* which is extended using four additional classes for the IIoT applications: *IIoTEndPoint, CloudEndPoint, ProcessEndPoint*, and *LogicalConnectionEndPoint*. The *IIoTEndPoint* contains special plugs to model Ethernet-based connectivity of IIoT based TCP/IP, MQTT, AMQP, RS 232, Modbus, and other communication available with the COTS target platform and device. The *CloudEndpoint* defines the interfaces for cloud communication such as the FTP and HTTP services, AMQP and MQTT for data-transfer. The *ProcessEndPoint* defines the traditional connectivity with non TCP/IP based protocols such as the Profibus using the gateway that delivers TCP/IP messages to the IIoT. The *LogicalConnectionEndpoints* model the communication between PLC and IIoT, master–slave, Bus, etc. In addition, we define *CommunicationRules* that enforce logically correct connectivity, e.g., MQTT to TCP/IP based devices. These interfaces model the physical interconnection of the components. The annotated AML CAEX schema thus generated

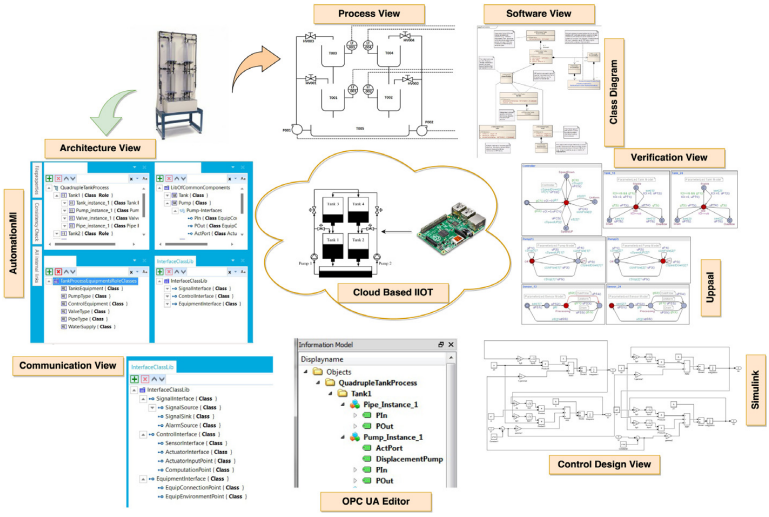


Fig. 2. Multi-view model of IIoT.

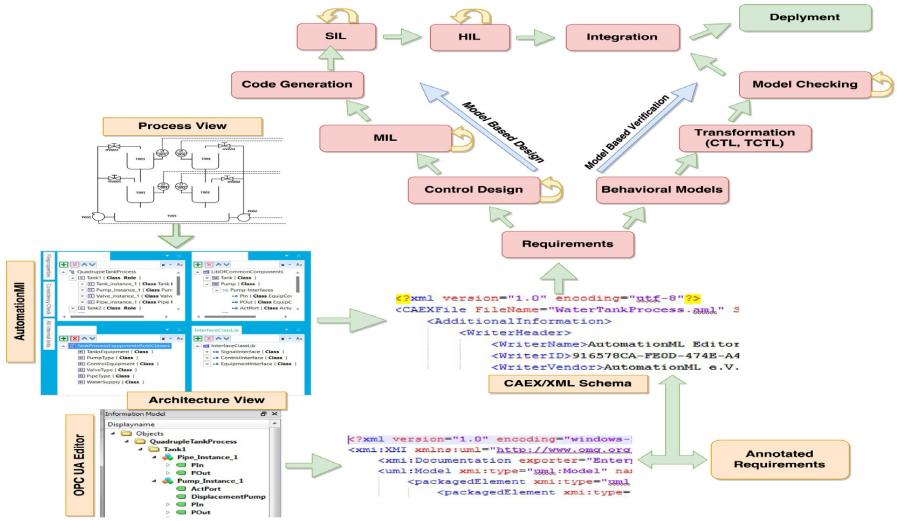


Fig. 3. Workflow for MBE based design of IIoT.

represents the multi-view model of the automation systems. The COTS target platform is mapped as resources to specific process control instances using UML models. The multi-view models and the annotated requirements are the input to the model-based design and verification steps.

The CAEX schema is then annotated with additional user-defined requirements either using UML models or textual representations. The requirements are generated for both the design and verification. The MBD and MBV approach used to design and verify automation systems will be detailed in the next sections. Of particular interest to this investigation is the design of MPC from the requirements. As MPC is emerging as a workhorse for smart manufacturing and one of the sophisticated control algorithms executed in process industry.

3.3. Model based design for cloud-based IIoT

The PLCs are the processing units for performing control in process industries and they are programmed using IEC 61131 standard. The role of MBD approach for auto-generating code for IEC 61131 based process automation has been studied in literature [14]. The PLCs are bit inflexible due to real-time requirements and sophisticated controllers such as MPCs are usually implemented on dedicated hardware platforms. Even in literature, the MPC implementation on PLC is quite scarce. With the emergence of IIoT, these sophisticated controllers can be realized in IIoT hardware and executing conventional control in traditional PLC systems. In this scenario, the MBD approach should be able to automate the code generation of sophisticated control schemes such as MPC. Therefore, for the rest of the section, we focus on the auto-code generation for MPC, rather than executing simple logics

or control actions such as Proportional Integral Derivative control. This brings down the cost and development time significantly.

The workflow for performing MBD-based design for cloud integrated IIoT is shown in Fig. 4 and it follows the V-model. It has four validation stages: Model-in-the-loop (MIL), software-in-the-loop (SIL), processor-in-the-loop (PIL) and Hardware-in-the-loop (HIL), before actual deployment. The real-time performance is validated through these different steps, an important requirement for process control applications. In the design flow requirements in the form of objective function (e.g., track a reference signal with minimum energy) and constraints (e.g., the maximum voltage of a pump) are fed as the requirements to the control design. The MPC parameters are computed based on the requirements using the design equations, (refer Appendix A for MPC models). Then both controller and process are simulated in a virtual environment to verify the control design, the procedure is called MIL. The model used is called platform independent model (PIM).

Following MIL, the target platform is identified, and the software code for the specific target is generated using an auto-coder. This model is called PDM, and the software emitted by the auto-coder is used to run the SIL, wherein the platform dependent software code and process models are simulated in virtual environment. This validation procedure tests the software code. The SIL code is ported to the target hardware, and tested on the virtual process with sensor and actuator models in the PIL validation, verifies the hardware capabilities, e.g., sampling time. Finally, the controller code working on the target hardware is interfaced to the sensors and it controls the virtual model of the process in the HIL. The validations are iterative procedures and design changes can be made based on the results. A controller design successfully validated in the four tests is deployed in the process industry with the control action performed by the target hardware. Two important observations here are:

1. There are not many auto-coders available for MPCs as they involve optimization solvers. These solvers face numerical accuracy, computational complexity and other numerical issues. This investigation used the jMPC,[2] a MATLAB based toolbox for auto-code generation for MPCs.
2. Combining the virtualization and multi-tenancy capabilities of cloud, when emulators of the specific hardware are in the cloud, then MBD can be performed from the cloud and the solution can be deployed in process industries.

3.4. Model based verification of IIoT

To perform model-based verification, the formal requirement specifications generated by AML (XML/CAEX schema) are mapped to abstract behavioural models (networks of timed automata). The automata model of the system under verification describes how the system is required to behave. The model, built in a suitable machine interpretable formalism is fed to model checker which verifies the model w.r.t properties of the specification. There are multiple different formalisms used for building formal requirement models. Our choice is Uppaal timed automata (UTA) [27] because the formalism is designed to express the timed behaviour of state transition systems and it has been previously been applied successfully to verify industrial automation systems in [28].

In the second step, the model templates for the component models are defined. The component models are modelled using UTA templates. The timed-action pattern shown in Fig. 5 is used to model the requirement specification following [29]. These are called the action patterns and timing wrapper have been presented

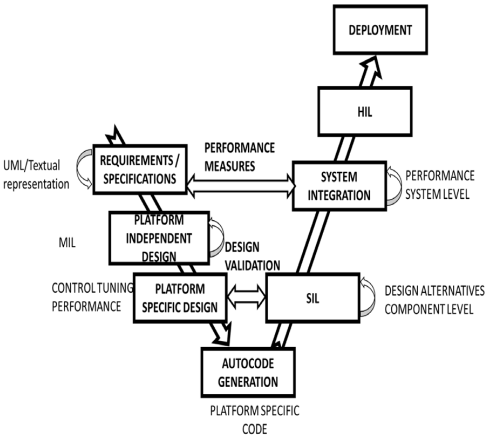


Fig. 4. MBD workflow.

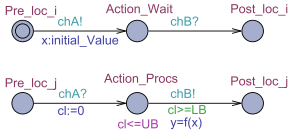


Fig. 5. A synchronous-parallel composition of time action pattern cf [29].

in [28] for industrial automation systems. The timing patterns are interlaced with the component models of the process industry, e.g., pump. The component models with their timing interfaces for a quadruple tank process is illustrated in Fig. 6. A detailed discussion on the models is presented in the results section.

In the third step, the model checker is used to verify the formal model w.r.t to a requirement specifications (properties). Like the model, the properties are expressed in a formal well-defined logics such as subset of CTL (computation tree logic) as in [29]. The CTL offers several temporal operators to express the requirements as CTL formulae can be classified by properties they express as reachability, safety and liveness, detailed analysis of these properties are provided in [29].

4. Cloud-intrinsic features for enabling flexibility in IIoT

This section highlights the opportunities in enhancing the performance of industrial automation by combining cloud capabilities with IIoT. In particular, three cases are considered: *(i)* plug-and-play control, *(ii)* smart manufacturing, and *(iii)* on-the fly verification.

4.1. Plug-and-play control

Vast control designs in industries are monolithic, i.e., entire control system needs to be changed, when a sub-system or hardware modifications are performed. As flexibility is emerging as a key requirement, control objectives of the plant change with time or even within production processes. In such scenarios, it is desirable to change control laws without diminishing existing controllers. The IIoT provides a way to flexibly change control algorithms using cloud services. The workflow for performing plug-and-play control is shown in Fig. 7. To guarantee security of the applications, the file transfers for the plug-and-play control happens using secure FTP, while for less important actions using FTP. In these scenarios, the

[2] http://www.i2c2.aut.ac.nz/Resources/Software/jMPCToolbox.html.

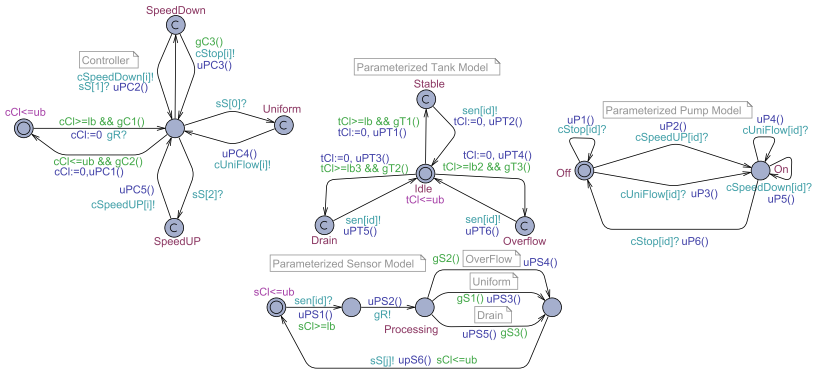


Fig. 6. Parameterized models of quadruple WaterTank process.

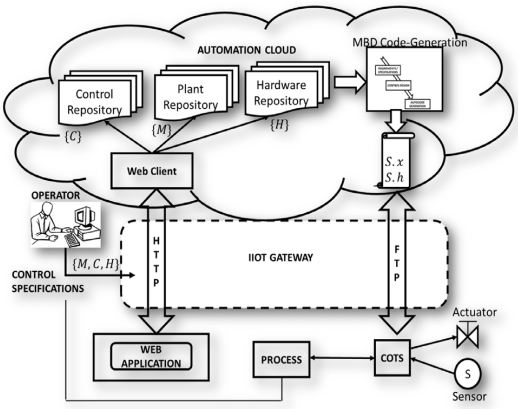


Fig. 7. MBD based plug-and-play control for IIoT.

user informs the cloud through a web interface about the changes. This is transmitted using HTTP interface of the IIoT gateway to the cloud. The cloud's HTTP interface receives this request. There are three components in the request, process, controller and hardware specification. The process model informs which of the process loop requires an upgrade, the controller instance/requirements, and the target hardware. The cloud then instantiate the virtual environment to obtain the process model, requirements for the specific controller, and controller design in PIM. It generates the PSM based on the controller specified by the user and ports it into the emulator and validates the design. Once the validation tests are successful, the control code is transmitted via IIoT gateway's FTP interface to the COTS embedded controller of the controlled process. Now, the control code is deployed on the hardware.

4.2. Smart manufacturing using cloud's auxiliary services

Computing power of IoT devices restricts their applications to perform computationally intensive task and is a major hindrance in the deployment of IIoT. Typically in smart manufacturing, data-mining models are used for creating knowledge from raw-data, both intrinsic and extrinsic to process industries. For example, forecasts on energy prices can be obtained using the data mining model and then integrated with optimization routines to perform smart manufacturing. Such data-mining models requires large memory for storing data and execution. They can also be available as third party applications as APIs. Exploiting the cloud features, the data-mining algorithms can be implemented in the cloud and knowledge aggregated can be transferred to the process controller using IIoT gateway using the MQTT and AMQP interfaces. Such aggregated knowledge can be embedded in the MPC controller for making knowledge based and optimization driven decisions.

4.3. On-the fly verification in the cloud

When hardware like sensor or actuators are updated, generally the control loop's timing performance is changed and the controller implementation needs to be modified as sampling and quantization levels have changed. If the devices match, such a scenario may not arise. But, the problem is faced with most legacy automation systems. When a sensor or actuator different from the one used is changed, the performance of the IIoT has to be verified. In our IIoT framework, the model templates (behaviour models) of the different components and their timing interfaces are available in the cloud's model repository. In case, a particular specification is unavailable for a model template, it is obtained from the field using a web application or FTP. These model templates are then composed and the MBV workflow is implemented from bottom to check whether the timing or safety requirements are met. This allows dynamic configuration of components in IIoT.

5. Results

5.1. Case study: Quadruple tank process

To illustrate the MBE approach for cloud-based IIoT, this investigation uses the quadruple tank process (QTP), a benchmark control problem in process control. The schematic of the QTP and its prototype used for deployment of IIoT is shown in Fig. 8. The QTP consists of four uniform sized cylindrical tanks with cross-sectional area A and outlet cross-sectional area a. In addition, there are two identical pumps namely Pump 1 and Pump 2. Four valves namely, HV 1, HV 2, HV 3 and HV 4 are provided to regulate the inlet liquid flow to the tanks. The objective of the low-level control is to maintain the liquid level in Tank 1 (h_1) and Tank 2 (h_2) at predefined value called the reference by varying the flow rate (f) of Pump 1 and Pump 2, by adjusting their supply voltage V_1 and V_2, respectively. The equations modelling the dynamics of QTP are given in Appendix B. To illustrate the use of the proposed approach, three use-cases are presented here: *(i)* Model-Based Design, *(ii)* plug-and-play control, and *(iii)* Model-Based Verification. The Raspberry PI 3 was chosen as the target hardware for our experiments.

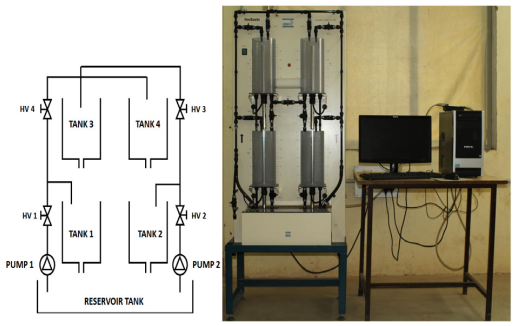


Fig. 8. Schematic and the process station of the quadruple tank process.

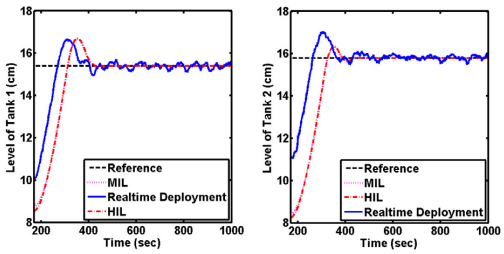


Fig. 9. MIL, HIL and SIL validation for M1.

5.2. Use-case: 1 model based deign

The MBD approach was used to design four different MPCs $\mathcal{M}_1 - \mathcal{M}_4$ described in Appendix A. The MBD workflow shown in Fig. was used to generate the auto-code using jMPC toolbox for the target embedded platform, Raspberry PI 3 in our case. The requirements were generated from the multi-view model of the QTP generated as shown in Fig. 2. The requirements are: offset-free tracking, faster response time (rise time) and settling time for the levels in the tank, i.e., h_1 and h_2. With MBD, the four MPC models $\mathcal{M}_1 - \mathcal{M}_4$ were studied. Our results showed that $\mathcal{M}_1$ met the design requirements and it was validated using MIL, HIL, and real-time deployment. The controller was deployed on the target hardware and it was used to control the process. The results of MIL, HIL and real-time deployment are shown in Fig. 9. While the MIL and HIL validated the results, small pulsations in the output are seen due to sensor noise from the environment that impacts the process performance. The other MPCs $\mathcal{M}_2$- $\mathcal{M}_4$ did not meet the requirements that were identified either during MIL, SIL or HIL. This results shows the ability of MBD approach to generate auto-code from requirements for even sophisticated controller such as MPC and to detect design issues early during the design phase, eliminating costly design upgrades later. It should be pointed here that using the emulator stack, the MBD approach can be performed in the cloud as well, thereby enabling cloud-based engineering of the solution.

5.3. Use-case:2 plug-and-play control

The user sends information on the requirements, hardware controller, and the process to the cloud. The cloud's virtualization and persistence services are used to generate the model templates for the process and controller as a PIM and the auto-code is emitted from the PSM for the target hardware. This is followed by SIL and the virtualization ability of the cloud is used to instantiate an emulator to perform the HIL. On successful validation of the requirements, the control code file is transferred using secure FTP interfaces of the cloud and IIoT gateway to the specific target platform. The plug-and-play deployment of the QTP is shown in Fig. 8 and the results obtained are shown in Fig. 10. One can verify that the plug-and-play control is performed and the requirements are met by the deployment. A slight pulsations are seen in the levels due to sensor's inertia and noise.

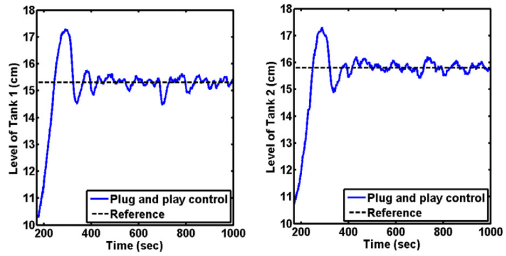


Fig. 10. Plug-and-play control in IIoT.

Table 1
Execution time of auto-generated code and CVXGEN code for the MPC models for 25 iterations.

MPC	Auto code (s)	CVXGEN code (s)
M1	24.8	198.77
M2	25	162.308
M3	24.9	113.24
M4	25	144.55

The computation time the plug-and-play control for the MPC models $\mathcal{M}_1 - \mathcal{M}_4$ is compared with the code generated by the auto-coder CVXGEN[3] for the target hardware within the process station (without file transfers). The computation times for 25 iterations of these codes are shown in Table 1. It can be seen that the auto-generated code for the target platform is lesser than CVXGEN code directly ported to the target hardware. This is due to run-time compilation that happens with the Python code as against compiled execution of the auto-generated code. This results demonstrates the plug-and-play capabilities introduced due to cloud's capabilities.

5.4. Use-case:3 model based verification

The UTA model for the quadruple water-tank process (QTP) is composed of automata of water tanks, sensors, pumps and controller are shown in Fig. 6. The model-templates using action model patterns and composition operators, that are used to construct the formal model of timing variations, and timing-wrapper is used in case of periodic operations. The composed model of the QTP with its component and timing interfaces is shown in Fig. 11.

5.4.1. Verification of requirement specifications

This investigation verifies QTP performance in two modes: minimum and non-minimum phase. In minimum phase mode, the level of *Tank 1* depends on the flow from *Pump1* and that of *Tank 2* is influenced by *Pump2* and this is a stable operation mode. While in non-minimum phase, the level of *Tank 1* depends on the flow from *Pump2* and that of *Tank 2* depends on *Pump1* leading to an unstable mode. To facilitate verification, the requirements specifications is mapped to the formal specifications of the QTP (for notations please refer the Nomenclature section).

[3] https://cvxgen.com/.

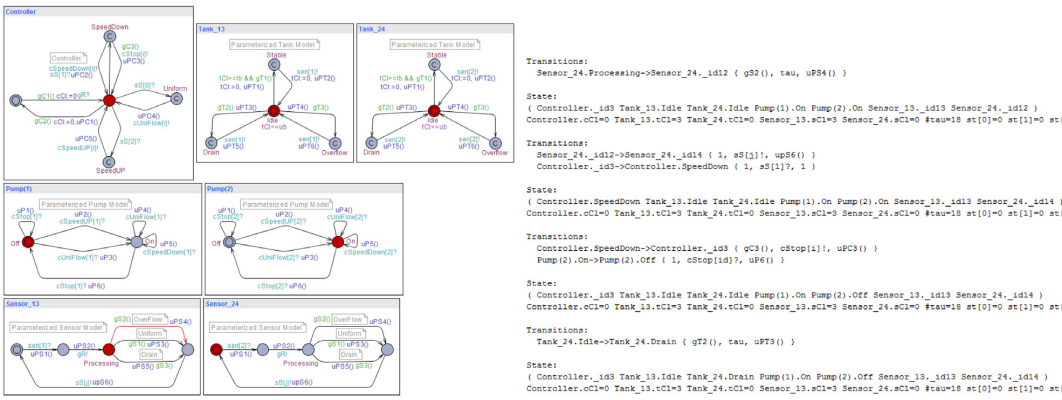


Fig. 11. Simulation and Generated Traces for above properties.

The level of the tank *w_Lev* and the additional parameter *TOver* are used to denote overflowing of the tank $w\_Lev >= TOver$, in such situations the pumps are slowed down. Including these new parameters, the UTA model is redefined for verifying the following properties:

- *(a) Deadlock Property*, we prove at first that there is not blocking states in the system. It is proved by running the model checking query
  $A[]$ *not deadlock*
- Verifying Minimum phase model of the QTP.
  Property 1: The reachability properties need to be verified for showing that the reaction time requirements are met. First we show that the both pumps supply sufficient water flow to tanks i.e.,
  $A <> Tank\_13.w\_Lev == TVol\_Max \&\& G\_Clock <= Ub$
  The query proves that the water tanks filling time from level 0 to *w_Lev* should not be exceed time bound *Ub*.
  Property 2: The property expresses that whenever the water level in particular tank reaches to *TOver* level, sensor measure the level and pass the signal to the controller, which issues the control signal *cStop* to the particular pump.
  $E <> Tank\_24.w\_Lev >= TOver\ imply\ (Tank\_24.Overflow \&\& p\_run[2] == 0 \&\& pCl <= Ub1)$
- Verification of non-minimum phase model of operation *Property 1:* Similarly as above, we prove that the controller issue the control signal to *Pump2* to maintain water level in *Tank13* as the requirement of Non-Minimum Phase Mode of Operation.
  $E <> Tank\_13.w\_Lev >= TOver\ imply\ (Controller.SpeedUP \&\& Pump(2).Off)$
  The query proves that the upon receiving signal from *sensor[1]* at *Tank13* (overflowing) the control issue a signal to *Pump2* to *Stop* or *speedDown* the water supply in *Tank3*.

The model checker generates the witness or counterexample depending upon if property is satisfied by the model. The automatic generation of witness and counterexample is considered as the key advantage of model checking which provides a useful source of diagnostic information and a basis for automated test generation. The Fig. 11 represents the simulation layout and generated traces for particular property. By using the model templates in the cloud, the MBV can be done on-the fly as illustrated in the example.

Comments: During the deployment of the MPCs in IoT devices, there were few issues that surfaced. First, the speed of the control algorithm depended on the target code language. For example, a C-code performed better than a run-time compiler language such as Python. Second, the latencies in the sensors and computations were not significant with on-board communications, but were significant in IP based communication. However, they were not at a level to destabilize the operations for the process application chosen. Third, the IoT controllers and sensors the effect of timing imperfections and noise created pulsations in the output. Fourth, there were some MPC implementations that could not be validated in the HIL, but they passed the other validation tests. Fifth, the real-time performance of the target platform is greatly influenced by the amount of TCP based communication used. Sixth, the cloud based communications and field level TCP communications generate only the same amount of latencies, this is partially due to the high computing power of the server. Finally, the cloud services communicating through the TCP based protocols have the same computation burden as any TCP device.

6. Conclusions

This investigation presented an IIoT architecture, a model-based engineering approach (MBE), and workflows for implementing cloud-based IIoT. The IIoT architecture combined the open connectivity with cloud-intrinsic features. To perform model based engineering, a multi-view model of the industrial automation capturing various aspects was proposed. A meta-model of the automation system integrating these different views was generated using AutomationML. This meta-model provided the basis for performing Model Based Engineering. Further, it generated the requirements for the design and verification. The Model Based Design (MBD) approach was used to design MPC, a sophisticated controller that repeatedly solves an optimization routine, for the target platform. The MBD approach generated the auto-code for the MPC and also validated the design through MIL, SIL and HIL during its workflow. The behaviour models from the requirements were used to perform model based verification. The Uppaal Timed Automata (UTA) models with action patterns of timing behaviour were composed to verify the timing performance to guarantee timing. Consequently, reducing the engineering efforts of cloud-based IIoT significantly. Insights into performing MBD and MBV from cloud was also provided. Next, the additional benefits provided by cloud-based IIoT was discussed with features such as plug-and-play control, smart manufacturing, and on-the-fly verification. The proposed IIoT architecture, MBE approach, and workflow were demonstrated on a QTP, a benchmark problem in process control. Our results showed the benefits of the combining cloud and IoT,

and MBE as an approach for realizing it. Studying deployment of cloud-based IIoT for providing enterprise wide connectivity and performing plant wide optimization are future course of this investigation.

Appendix

MPC optimization models

In the MBD workflow, the objective function and constraints of the MPC denote the requirements of the control algorithm. The investigation considers four different MPC models, they are:

$$\begin{aligned}
\mathcal{M}_1 &: \underset{U}{\text{minimize}}\ \mathrm{J} = (Y - Y_r)^T Q (Y - Y_r) + \Delta U^T R \Delta U \\
&\text{Subject to: } \mathcal{C} \\
\mathcal{M}_2 &: \underset{U}{\text{minimize}}\ \mathrm{J} = (Y - Y_r)^T Q (Y - Y_r) + \\
&\qquad (U - u_d)^T R (U - u_d) \\
&\text{Subject to: } \mathcal{C} \\
\mathcal{M}_3 &: \underset{U}{\text{minimize}}\ \mathrm{J} = (Y - Y_r)^T Q (Y - Y_r) + U^T R U \\
&\text{Subject to: } \mathcal{C} \\
\mathcal{M}_4 &: \underset{U}{\text{minimize}}\ \mathrm{J} = (Y - Y_r)^T Q (Y - Y_r) + \Delta U^T R \Delta U \\
&\text{Subject to: } \mathcal{C}
\end{aligned} \tag{1}$$

where the constraints $\mathcal{C}$ is given by

$$\begin{aligned}
x(k+1) &= Ax(k) + Bu(k) + \hat{d}(k), \quad \forall k = 1, ..N_p \\
y(k) &= Cx(k) \ \forall k = 1, ..N_p \\
u_{min} &\le u(k) \le u_{max} \ \ \forall k = 1, ..N_p \\
\Delta u_{min} &\le \Delta u(k) \le \Delta u_{max} \ \ \forall k = 1, ..N_p \\
y_{min} &\le y(k) \le y_{max} \ \ \forall k = 1, ..N_p
\end{aligned}$$

These constraints model the physical and operating constraints of the MPC. They capture the system dynamics, constraints on the control input, change in control input and output, respectively.

Quadruple tank process dynamics

The dynamics of the quadruple process is given by

$$\begin{aligned}
\dot{h}_1(t) &= \frac{1}{A}(a\sqrt{2gh_3} + \gamma_1 f_1 - a\sqrt{2gh_1}) \\
\dot{h}_2(t) &= \frac{1}{A}(a\sqrt{2gh_4} + \gamma_2 f_2 - a\sqrt{2gh_2}) \\
\dot{h}_3(t) &= \frac{1}{A}((1-\gamma_2) f_2 - a\sqrt{2gh_3}) \\
\dot{h}_4(t) &= \frac{1}{A}((1-\gamma_1) f_2 - a\sqrt{2gh_4})
\end{aligned} \tag{2}$$

References

[1] C. Wang, Z. Bi, L. Da Xu, IoT and cloud computing in automation of assembly modeling systems, IEEE Trans. Ind. Inf. 10 (2) (2014) 1426–1434.
[2] A. Botta, W. De Donato, V. Persico, A. Pescapé, Integration of cloud computing and internet of things: a survey, Future Gener. Comput. Syst. 56 (2016) 684–700.
[3] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of things (IoT): A vision, architectural elements, and future directions, Future Gener. Comput. Syst. 29 (7) (2013) 1645–1660.
[4] A.W. Colombo, S. Karnouskos, O. Kaynak, Y. Shi, S. Yin, Industrial cyber-physical systems: A Backbone of the Fourth Industrial Revolution, IEEE Ind. Electron. Mag. 11 (1) (2017) 6–16.
[5] W. He, L. Da Xu, Integration of distributed enterprise applications: A survey, IEEE Trans. Ind. Inf. 10 (1) (2014) 35–42.
[6] L. Da Xu, W. He, S. Li, Internet of things in industries: A survey, IEEE Trans. Ind. Inf. 10 (4) (2014) 2233–2243.
[7] I. Calvo, F. Pérez, I. Etxeberria, G. Morán, Control communications with DDS using IEC61499 service interface function blocks, in: Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on, IEEE, 2010, pp. 1–4.
[8] H.M. Hanisch, Closed-loop modeling and related problems of embedded control systems in engineering, in: International Workshop on Abstract State Machines, Springer, 2004, pp. 6–19.
[9] S. Theiss, V. Vasyutynskyy, K. Kabitzsch, Software agents in industry: A customized framework in theory and praxis, IEEE Trans. Ind. Inf. 5 (2) (2009) 147–156.
[10] F. Jammes, H. Smit, Service-oriented paradigms in industrial automation, IEEE Trans. Ind. Inf. 1 (1) (2005) 62–70.
[11] V.N. Dubinin, V. Vyatkin, Semantics-robust design patterns for IEC 61499, IEEE Trans. Ind. Inf. 8 (2) (2012) 279–290.
[12] K. Thramboulidis, Model-integrated mechatronics-toward a new paradigm in the development of manufacturing systems, IEEE Trans. Ind. Inf. 1 (1) (2005) 54–61.
[13] V. Vyatkin, Software engineering in industrial automation: state-of-the-art review, IEEE Trans. Ind. Inf. 9 (3) (2013) 1234–1249.
[14] M. Obermeier, S. Braun, B. Vogel-Heuser, A model-driven approach on object-oriented PLC programming for manufacturing systems with regard to usability, IEEE Trans. Ind. Inf. 11 (3) (2015) 790–800.
[15] B. Vogel-Heuser, D. Schütz, T. Frank, C. Legat, Model-driven engineering of manufacturing automation software projects–a SysML-based approach, Mechatronics 24 (7) (2014) 883–897.
[16] K. Thramboulidis, F. Christoulakis, UML4IoT-A UML-based approach to exploit IoT in cyber-physical manufacturing systems, Comput. Ind. 82 (2016) 259–272.
[17] S. Mayer, J. Hodges, D. Yu, M. Kritzler, F. Michahelles, An open semantic framework for the industrial internet of things, IEEE Intell. Syst. 32 (1) (2017) 96–101.
[18] F. Tao, Y. Cheng, L. Da Xu, L. Zhang, B.H. Li, CCIoT-CMfg: cloud computing and internet of things-based cloud manufacturing service system, IEEE Trans. Ind. Inf. 10 (2) (2014) 1435–1442.
[19] Z. Bi, L. Da Xu, C. Wang, Internet of things for enterprise systems of modern manufacturing, IEEE Trans. Ind. Inf. 10 (2) (2014) 1537–1546.
[20] A.W. Colombo, T. Bangemann, S. Karnouskos, IMC-AESOP outcomes: Paving the way to collaborative manufacturing systems, in: Industrial Informatics (INDIN), 2014 12th IEEE International Conference on, IEEE, 2014, pp. 255–260.
[21] A.W. Colombo, T. Bangemann, S. Karnouskos, J. Delsing, P. Stluka, R. Harrison, F. Jammes, J.L. Lastra, et al., Industrial cloud-based cyber-physical systems, in: The IMC-AESOP Approach, Springer, 2014.
[22] A.W. Colombo, S. Karnouskos, J.M. Mendes, P. Leitão, Industrial agents in the era of service oriented architectures and cloud based industrial infrastructures, Ind. Agents: Emerging Appl. Softw. Agents Ind. (2015) 67–87.
[23] A. Maffei, S. Srinivasan, P. Castillejo, J.F. Martinez, L. Iannelli, E. Bjerkan, L. Glielmo, A semantic middleware supported receding horizon optimal power flow in energy grids, IEEE Trans. Ind. Inf. (2017).
[24] K. Thramboulidis, et al., The 3+ 1 SysML view-model in model integrated mechatronics, J. Softw. Eng. Appl. 3 (02) (2010) 109.
[25] J. Reineke, S. Tripakis, Basic problems in multi-view modeling, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2014, pp. 217–232.
[26] B. Brandenbourger, M. Vathoopan, A. Zoitl, Engineering of automation systems using a metamodel implemented in automationml, in: Industrial Informatics (INDIN), 2016 IEEE 14th International Conference on, IEEE, 2016, pp. 363–370.
[27] S. Balasubramaniyan, S. Srinivasan, F. Buonopane, B. Subathra, J. Vain, S. Ramaswamy, Design and verification of cyber-physical systems using truetime, evolutionary optimization and UPPAAL, Microprocess. Microsyst. 42 (2016) 37–48.
[28] S. Srinivasan, F. Buonopane, J. Vain, S. Ramaswamy, Model checking response times in networked automation systems using jitter bounds, Comput. Ind. 74 (2015) 186–200.
[29] S. Srinivasan, F. Buonopane, S. Ramaswamy, J. Vain, Verifying response times in networked automation systems using jitter bounds, in: Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, IEEE, 2014, pp. 47–50.

Muthukumar Natarajan is currently pursuing his Ph.D. in SASTRA University from 2014. His research interests are broader topics of optimization driven control, the Internet of Things (IoT), and process control. He is a recipient of the SASTRA research scholarship for pursuing Ph.D. and had a brief stint with industry before embarking on the current position. He has 8 journal papers and 10 conference papers published through his research work.

Seshadhri Sriniavsan obtained his Ph.D. from National Institute of Technology-Tiruchirappalli in 2010. He is currently working with Berkeley Education Alliance for Research in Singapore (BEARS), Singapore working with Prof. Kameshwar Poolla. He is working on SinBerBEST2 and GBIC projects. Before taking up this position, he was a researcher at GRACE, Italy, Technical University of Munich, Germany, Kalasalingam Academy of Research and Education, and Center for Excellence in Nonlinear Systems. He worked in European projects eGotham, I3RES, and Complex networked control systems. In 2011, he worked as an Assoc. Scientist with ABB Global Industries and Services Pvt. Ltd., Indian Corporate Research Center, India. He was involved with the IEEE CSS Standardization committee in 2016 and was awarded the IEEE CSS Outreach fund in 2017. In addition, he has teaching stints at Kalasalingam Academy of Research and Education. His publications include around 50 journals, reputed conferences, 4 books, and has patented 3 technologies.

Kannan Ramkumar was born in Madurai, India, in 1975. He received the B. Tech degree in Instrumentation and control Engineering from Madurai Kamaraj University in 1997, subsequently he did his M. Tech from Regional Engineering college, Trichy in 2000 and obtained his Ph.D. degree in Control Engineering from SASTRA Deemed University, India in 2010. Since 1998, he has been with the Department of Electronics and Instrumentation, SASTRA Deemed University, where he was an Assistant Professor, became an Associate Professor in 2011, and a Professor in 2018. His current research interests include Mobile Robotics, Estimation and Control theory and Electrical drive systems. He has received funding from DRDO, an Indian government defense agency, for investigating the localization and mapping problem in Mobile robot. He was the Chair Professor for the Wipro Mission 10X, a pedagogy skills improvement initiative in SASTRA University for the year 2011–12. He is the recipient of the prestigious "Innovative Practitioner" a teaching excellence award from Wipro for the year 2011.Currently he is solving Machine learning, Estimation and Control problems related to applications like Electric vehicle, Mobile robots and Process control. He is also heading CALIBRE (Control Artificial intelligence, Biomedical and Robotics Engineering), a research group.

Deepak Pal is currently pursuing his Ph.D. in the department of software science, Tallinn University of technology, Tallinn, Estonia. Currently, he is studying and implementing the problem of generating tests for real-time distributed systems by exploiting model-based techniques and has publication in reputed conferences. His research interests are software engineering, software testing, model-based testing, model checking, theorem proving, program verification, modelling, real-time distributed systems, cyber–physical systems.

Jüri Vain graduated in System Engineering from Tallinn Polytechnic Institute, Estonia in 1979. He received his Ph.D. in computer science from the Institute of Cybernetics at Estonian Academy of Sciences in 1987. Currently, he is Professor of Computer Science at the Department of Software Science, Tallinn University of Technology. His research interests include formal methods, model-based testing, cyber–physical systems, human–computer interaction, autonomous robotics, and artificial intelligence. He has been leading researcher in several international projects under EU framework programmes and the coordinator of Estonian-Japan medical robotics collaboration programme. He has published over 150 scientific articles including journal papers, book chapters and conference papers.

Srini Ramaswamy currently serves as a Global R&D Project Manager for Engineering in the Power Generation Business Unit. Earlier he was the *global lead for Software Tools Development and Services* for the Software Development Improvement Program (SDIP), *headed the Industrial Software Systems research* group at its India Corporate Research Center (CRC), and headed the *Tools and Support Services* group for its India Development Center (IDC) in Bangalore, India. On the academic front, he serves as a visiting professor at the University of Arkansas at Little Rock and as the Associate Director for the Australia–India Centre for Automation Software Engineering, a $3M three-way Industry–University–Government partnership focused on software systems research for the Engineering and Automation sectors. He is on the Executive Committee of the Computing Accreditation Commission of ABET, the IEEE-CS representative member of the board at CSAB, a senior member of the ACM, and a senior member of the IEEE.

Curriculum Vitae

1. Personal data

Name	Deepak Pal
Date and place of birth	12 January 1989, Meerut, Uttar Pradesh, India
Nationality	Indian

2. Contact information

Address	Käokella tee 1a, Järveküla 75304 Harju maakond, Estonia
Phone	+372 56828921
E-mail	deepak.pal@ttu.ee

3. Education

2014–2020	Tallinn University of Technology, Department of Software Science, Computer Science, PhD studies
2013–2014	Tallinn University of Technology, Department of Software Science, Informatics, MSc *Exchange Program*
2012–2013	National Institute of Technology, Rourkela India, Department of Computer Science, Computer Science, MSc

4. Language competence

Hindi	Native
English	C2
Estonain	B1

5. Professional employment

2017– . . .	SEB Bank AS, Full Stack Developer

6. Honours and awards

- 2013, I got selected for scholarship under the HERITAGE Eramus Mundus Action 2 project, on a masters mobility program at Tallinna Tehnikaulikool.
- 2014, IT Academy Skype Scholarship.
- 2015, European Innovation Academy Scholarship.
- 2015, Summer School Marktoberdorf 2015, Verification and Synthesis of Correct and Secure Systems. Received a scholarship for attending International Summer School Marktoberdorf 2015 Sponsors by NATO Science for peace and security program.

7. Defended theses

- 2013, Model Based Conformance Testing of Distributed Systems, MSc, supervisor Professor Santanu Rath (National Institute of Technology, Rourkela, Orissa, India) and Professor Jüri Vain, Tallinn University of Technology, Department of Software Science, Tallinn, Estonia.

8. Field of research

- Software Engineering, Software Testing, Formal Methods, Model Based Testing, Model checking, Theorem proving, Real-time Distributed Systems, Cyber Physical Systems.

Papers

1. Pal, Deepak; Vain, Jüri (2019). Model based test framework for communications-critical internet of things systems. Databases and Information Systems X: Selected Papers from the Thirteenth International Baltic Conference, DB&IS 2018. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Amsterdam: IOS Press, 79-94

2. Pal, Deepak; Vain, Jüri (2019). A systematic approach on model refinement and regression testing of real-time distributed systems. 9th IFAC Conference on Manufacturing Modelling, Management and Control, MIM 2019 : Berlin, Germany, 28-30 August 2019, Proceedings. Ed. Ivanov, Dmitry; Dolgui, Alexandre; Yalaoui, Farouk. Elsevier, 1091-1096

3. Pal, Deepak; Vain, Jüri (2018). Model based approach for testing: distributed real-time systems augmented with online monitors. Databases and Information Systems : 13th International Baltic Conference, DB&IS 2018, Trakai, Lithuania, July 1-4, 2018, Proceedings. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Cham: Springer, 142-157

4. Pal, Deepak; Vain Jüri; Srinivasan, Seshadhri; Ramaswamy, Srini (2017). Model-based maintenance scheduling in flexible modular automation systems. 22nd IEEE International Conference on Emerging Technologies and Factory Automation, EFTA' 2017 : September 12-15, 2017, Limassol, Cyprus, 1-6

5. Pal, D.; Vain, J. (2016). Generating optimal test cases for real-time systems using DIVINE model checker. BEC 2016 : 15th Biennial Baltic Electronics Conference, Tallinn University of Technology, October 3-5, 2016 Tallinn, Estonia, 99–102

6. Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Automatic Distribution of Local Testers for Testing Distributed Systems. In: Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. (Ed.). Databases and Information Systems ix: Selected Papers from the Twelfth International Baltic Conference, DB&IS 2016 (297-310). Amsterdam: IOS Press. (Frontiers in Artificial Intelligence and Applications; 291)

7. Muthukumar, N.; Srinivasan, Seshadhri; Ramkumar, K.; Pal, Deepak; Vain, Jüri; Ramaswamy, Srini (2019). A model-based approach for design and verification of Industrial Internet of Things. Future Generation Computer Systems, 354–363

Elulookirjeldus

1. Isikuandmed

Nimi	Deepak Pal
Sünniaeg ja -koht	12 Jaanuar 1989, Meerut, Uttar Pradesh, India
Kodakondsus	India

2. Kontaktandmed

Aadress	Käokella tee 1a, Järveküla 75304 Harju maakond, Eesti
Phone	+372 56828921
E-mail	deepak.pal@ttu.ee

3. Haridus

2014-2020	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Info- ja kommunikatsioonitehnoloogia, doktoriõpe
2013–2014	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Informaatika, MSc
2012–2013	National Institute of Technology, Rourkela India, Department of Computer Science, Computer Science, MSc

4. Keelteoskus

Hindi	Emakeel
Inglise keel	C2
Eesti keel	B1

5. Teenistuskäik

2017– . . .	SEB Bank AS, tarkvara arendaja

6. Autasud

- 2013, HERITAGE Eramus Mundus Action 2 project, masters mobility at Tallinna Tehnikaülikool.
- 2014, IT-Akadeemia Skype Scholarship.
- 2015, European Innovation Academy grant.
- 2015, Marktoberdorfi suvekooli “Verification and Synthesis of Correct and Secure Systems” (sponsors by NATO Science for peace and security program) osalemisgrant

7. Kaitstud lõputööd

- 2013, Hajussüsteemide mudeli-põhine konformsustestimine, MSc, juhendaja Professor Santanu Rath (National Institute of Technology, Rourkela, Orissa, India) ja Professor Jüri Vain, Tallinna Tehnikaülikool, Arvutiteaduse instituut.

8. Teadustöö põhisuunad

- Tarkvaratehnika, tarkvara testimine, formaalmeetodid, mudelipõhine testimine, mudelkontroll, teoreemitõestamine, reaalaja hajussüsteemid, küber-füüsikalised süsteemid.

Papers

1. Pal, Deepak; Vain, Jüri (2019). Model based test framework for communications-critical internet of things systems. Databases and Information Systems X: Selected Papers from the Thirteenth International Baltic Conference, DB&IS 2018. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Amsterdam: IOS Press, 79-94

2. Pal, Deepak; Vain, Jüri (2019). A systematic approach on model refinement and regression testing of real-time distributed systems. 9th IFAC Conference on Manufacturing Modelling, Management and Control, MIM 2019 : Berlin, Germany, 28-30 August 2019, Proceedings. Ed. Ivanov, Dmitry; Dolgui, Alexandre; Yalaoui, Farouk. Elsevier, 1091-1096

3. Pal, Deepak; Vain, Jüri (2018). Model based approach for testing: distributed real-time systems augmented with online monitors. Databases and Information Systems : 13th International Baltic Conference, DB&IS 2018, Trakai, Lithuania, July 1-4, 2018, Proceedings. Ed. Lupeikiene, Audrone; Vasilecas, Olegas; Dzemyda, Gintautas. Cham: Springer, 142-157

4. Pal, Deepak; Vain Jüri; Srinivasan, Seshadhri; Ramaswamy, Srini (2017). Model-based maintenance scheduling in flexible modular automation systems. 22nd IEEE International Conference on Emerging Technologies and Factory Automation, EFTA' 2017 : September 12-15, 2017, Limassol, Cyprus, 1-6

5. Pal, D.; Vain, J. (2016). Generating optimal test cases for real-time systems using DIVINE model checker. BEC 2016 : 15th Biennial Baltic Electronics Conference, Tallinn University of Technology, October 3-5, 2016 Tallinn, Estonia, 99–102

6. Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Automatic Distribution of Local Testers for Testing Distributed Systems. In: Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. (Ed.). Databases and Information Systems ix: Selected Papers from the Twelfth International Baltic Conference, DB&IS 2016 (297-310). Amsterdam: IOS Press. (Frontiers in Artificial Intelligence and Applications; 291)

7. Muthukumar, N.; Srinivasan, Seshadhri; Ramkumar, K.; Pal, Deepak; Vain, Jüri; Ramaswamy, Srini (2019). A model-based approach for design and verification of Industrial Internet of Things. Future Generation Computer Systems, 354–363