Proceedings of the Estonian Academy of Sciences,
2017, **66**, 1, 75–88
https://doi.org/10.3176/proc.2017.1.08
Available online at www.eap.ee/proceedings

COMPUTER SCIENCE

DTRON: a tool for distributed model-based testing of time critical applications

Aivo Anier, Jüri Vain, and Leonidas Tsiopoulos*

Tallinn University of Technology, Ehitajate tee 5, 12616 Tallinn, Estonia

Received 27 June 2016, accepted 1 December 2016, available online 2 March 2017

Abstract. Cyber-Physical Systems (CPS) present the greatest challenges but also the greatest opportunities in several critical industrial segments such as electronics, automotive and industrial automation. Governing the complexity and design correctness issues of CPS software requires methodologies and tools that address the problems of intrinsic concurrency and timing constraints over a large spectrum of CPS architectures. In this paper we present DTRON, a framework for model-based testing that addresses the issues of distributed execution and real-time constraints imposed by the design of networked CPS. DTRON extends the Uppaal model checking tool and online test execution tool TRON enabling coordination, synchronization, and online distributed testing. The notion of Δ-testability required to guarantee the controllability of distributed tests is one of the main design considerations for DTRON. The core part of the paper presents the architectural solutions for implementing DTRON and then special focus is put on the performance evaluation of the tool taking into account the communication and test adapter delays in networked systems. We demonstrate that the co-use of Spread message serialization service and Network Time Protocol allows reducing Δ down to the 1 ms range, which is sufficient for testing timing properties of a substantial class of networked CPS. We exemplify the applicability of DTRON with three distributed testing case studies, namely, city street light controller network, interbank trading system, and robot navigation system.

Key words: computer science, formal methods, model-based testing, distributed systems, real-time systems, cyber-physical systems, Uppaal timed automata.

1. INTRODUCTION

Cyber-Physical Systems (CPS) combine a cyber side (computing and networking) with a physical side (mechanical, electrical, and chemical processes). Contemporary CPS often grow to the scale of global geographic distribution and latency requirements are measured in nanoseconds. Governing the complexity and design correctness issues of CPS software requires major advancement in models, algorithms, methods, and tools that will incorporate verification and validation of software and systems at the design stage [1]. In particular, the problems of intrinsic concurrency and timing constraints over a wide spectrum of CPS heterogeneous architectures need to be addressed. Among other design related issues the time criticality where the reaction time is a primary design consideration is a serious challenge also to existing integration and/or system level testing techniques and tools. Although state-of-the-art testing tools support the prescribed input profiles, they seldom provide enough reactivity to run the tests with simultaneous and interdependent input profiles at remote frontends. The complexities 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.

Recently David et al. [2] proposed explicit test controllability criteria, the so-called Δ-testability criteria, for model-based remote online testing where Δ denotes an upper bound of message propagation delay between the remote tester and the ports of the System Under Test (SUT). For generating test stimuli based on SUT outputs the tester needs to wait the reactions to earlier inputs

* Corresponding author, leonidas.tsiopoulos@ttu.ee

generated and sent to the SUT. So, the input stimuli can reach the SUT earliest in 2Δ after preceding outputs have occurred on SUT ports. This is the theoretical minimum reaction time the tester can account on to guarantee the controllability of tests. However, even this condition makes remote testing problematic when 2Δ is close to timing constraints to be tested and the delay of generating test stimuli within test adapters is non-negligible. For example, in CPS such as traffic control systems it is common that the time window of receiving inputs at various ports has a definite effect on the system reaction. Thus, to achieve the test coverage of such behaviours under study, the test deployed on the test execution environment should strictly guarantee that the actual time interval between receiving a SUT output and sending the subsequent test stimulus satisfies these timing conditions.

The theoretical foundations of testing time critical distributed systems have been presented in several papers, for example in [3–6]. Tools such as Uppaal TRON [7], TTG [8], etc. have been developed to support online testing of time critical systems locally, in a non-distributed manner. To the best of our knowledge, very few attempts have been made to develop proper tool support with the necessary synchronizations/coordinations between distributed local testers for testing distributed (hard) real-time systems. Ru Dai et al. [9] presented TimedTTCN-3, a real-time extension for TTCN-3 (Testing and Test Control Notation) for testing the functional behaviour of distributed systems. The framework does not adhere to the notion of Model-Based Testing (MBT) where the test cases are generated from formally specified models and it is assumed that the necessary tester synchronizations would be addressed by the test execution environment without presenting further details on this important issue. The commercial tool TestComplete by SmartBear [10] targets distributed testing for websites and server applications when they work with multiple clients simultaneously. An attempt to extend the existing theoretical results by Khoumsi [6] with an actual tool implementation for overcoming synchronization issues between local testers and their attached clocks is presented in [11] where the IEEE 1588 Precision Time Protocol is partly implemented. The implementation targeted strictly a specific multicomputer system with PowerPC computing nodes interconnected with a RACEway Interconnection through sockets. The tool is non-existent currently and its applicability for networked real-time applications over the internet cannot be assessed. Hence, the proper tool support for distributed Δ-testing of critical real-time systems is still missing.

In this paper we present DTRON [12], an execution framework for distributed model-based testing that addresses the issues of concurrency, scalability, and real-time constraints imposed by the design of networked CPS. DTRON is one of the first attempts to merge timing with specifics of remote and distributed testing. It relies on the Uppaal model checking tool [13] and on-line test execution tool Uppaal TRON. The Uppaal tool includes a model editor, simulator, and model checking engine for Timed Computation Tree Logic (TCTL). Uppaal TRON is a local testing tool, based on the Uppaal engine, suited for black-box conformance testing of timed systems, mainly targeted for embedded software. DTRON extends these tools by enabling coordination and synchronization of distributed tester components.

The core part of the paper focuses on the performance characteristics of DTRON in distributed MBT taking into account the communication and additional processing delays in networked systems. The relevance for Δ-testing that is required to guarantee the controllability of distributed tests, is one of the main design concerns for DTRON studied in this paper. After the presentation of the required preliminaries for DTRON, the key solutions of DTRON's architectural design are introduced. Special focus is put on the performance evaluation of the tool taking into account the communication and test adapter delays in networked configurations. We demonstrate that the co-use of Spread message serialization service and Network Time Protocol allows reducing Δ down to the 1 ms range, which is sufficient for testing timing properties of a substantial class of networked CPS. We exemplify the usability of DTRON with three distributed testing case studies, namely, a city street lighting control network, interbank trading and stock exchange system, and Robot Operating System (ROS)-based navigation system.

2. PRELIMINARIES

2.1. Model-based testing

DTRON is one component in the provably correct model-based design (MBD) tool chain [14]. The emphasis in this paper is put on online distributed MBT, although DTRON is also applicable as an execution platform for distributed model-based control applications [15]. In this paper, we interpret MBT in the standard way, i.e. as conformance testing that compares the expected behaviours described by the system requirements model with the observed behaviours of an actual implementation of the SUT. In MBT the development is manifested in five main steps. Step 1 is about the modelling of the SUT. Step 2 concerns the specification of the testing goal applied in different forms: a set of test scenarios, safety constraints to be followed, the target state of the SUT to be reached, etc. All of these goals need to be expressed in terms of the SUT model elements and their attributes. In Step 3 a set of tests (test suite) is generated to reach the testing goal and in Step 4 the adapters that interface models with the SUT are implemented. In Step 5 the executable test configuration is deployed on the SUT and executed.

Scalable integration and system level testing of networked CPS require complex tools and techniques to assure a good quality of the test results. To achieve trustworthy test results, the generated test suite needs to be verified and its execution tool validated. Therefore, the development process of DTRON executable tests considers development phases paired with verification and timing correctness assurance steps. The formal properties verified in the course of modelling the SUT are *connectedness of the model control graph*, *output observability*, *input enabledness*, the *absence of deadlocks*, and *strong responsiveness*. Besides these, the application specific correctness properties are verified by model checking and static analysis. Additionally, simulation with the Uppaal tool is used for visual inspection of behaviours and for detection of inconsistencies between the model and the system described. These measures cannot fully guarantee the absolute correctness of the SUT models, but they provide confidence that the properties needed for automatic test generation from the SUT model are satisfied.

Another usability issue is uniformness of methods and tools needed to implement different steps of the development processes. It means covering the SUT model construction, test goal specification, tester synthesis, test adapter building, and deployed test configuration execution steps by the same tool set. Along with the Uppaal family toolset, DTRON focuses on specific development steps such as SUT *adjustment* for defining test interfaces, *test deployment*, and *execution*.

While SUT modelling is mostly formalizing SUT design requirements and the test purpose, the deployment and model-based execution are the steps where beside formal semantics of models also real world constraints need to be taken into account, e.g. test components, computational and communication deployment architecture, how the test i/o alphabet is mapped to test interfaces, scheduling policy, software/hardware jitter, and implementation imperfections. The main goal in designing the deployment architecture and execution environment is to achieve that the physical test execution follows the SUT model defined semantics as much as possible. The conceptual execution architecture of MBT is shown in Fig. 1. The SUT is first manipulated and its reaction is observed by a *Test Adapter*; the *Monitor* (test oracle) estimates if the observed state of the SUT conforms to the state predicted by the model; *TestSpec* comprises the model of implementation and its environment. Testing goals are stated by *Coverage criteria/facts*, and the *Selector* is responsible for guiding the test case selection and test execution functionality.

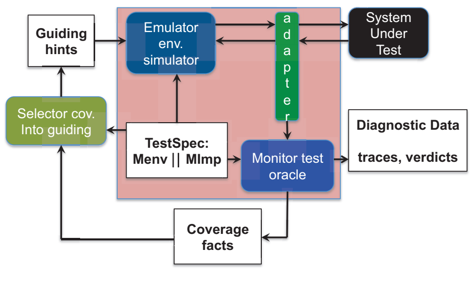


Fig. 1. Conceptual view of the DTRON model-based testing architecture.

2.2. Uppaal Timed Automata

Uppaal Timed Automata (UTA) [16] is the formalism of choice for modelling and test case generation for DTRON because it is expressive enough to support the real-time constraints modelling and the decision procedures are efficient enough to support the practical verification. Also, 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 into account further motivates the choice for UTA.

UTA 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. A UTA is given as the tuple $(L, E, V, CL, Init, Inv, T_L)$, where L is a finite set of locations; E is the set of edges defined by $E \subseteq L \times G(CL,V) \times Sync \times Act \times L$, where $G(CL,V)$ is the set of constraints allowed in guards, $Sync$ is a set of synchronization actions over channels. An action *send* over a channel h is denoted by $h!$ and its co-action *receive* is denoted by $h?$. Act is a set of sequences of assignment actions with integer and boolean expressions as well as with clock resets. V denotes the set of integer and boolean variables. CL denotes the set of real-valued clocks $(CL \cap V = \varnothing)$. $Init \subseteq Act$ is a set of assignments that assigns the initial values to variables and clocks. $Inv : L \to I(CL,V)$ is a function that assigns an invariant to each location, $I(CL,V)$ is the set of invariants over clocks CL and variables V. $\mathrm{T}_L : L \to \{ordinary, urgent, committed\}$ is the function that assigns the type to each location of the automaton.

We introduce the semantics of UTA as defined in [16]. A clock valuation is a function $val_{cl} : CL \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. A variable valuation is a function $val_v : V \to \mathbb{Z} \cup BOOL$ from the set of variables to integers and booleans. Let $\mathbb{R}^{CL}$ and $(\mathbb{Z} \cup BOOL)^V$ be the sets of all clock and variable valuations, respectively. The semantics of a UTA is defined as a Labelled Transition System (LTS) $(\Sigma, init, \rightarrow)$, where $\Sigma \subseteq L \times \mathbb{R}^{Cl} \times (\mathbb{Z} \cup BOOL)^V$ is the set of states, the initial state $init = Init(cl, v)$ for all $cl \in CL$ and for all $v \in V$, with $cl = 0$, and $\rightarrow \subseteq \Sigma \times \{\mathbb{R}_{\geq 0} \cup Act\} \times \Sigma$ is the transition relation such that:

- $(l, val_{cl}, val_v) \xrightarrow{\mathtt{d}} (l, val_{cl} + \mathtt{d}, val_v)$ if $\forall \mathtt{d}' : 0 \leq \mathtt{d}' \leq \mathtt{d} \Rightarrow val_{cl} + \mathtt{d}' \models Inv(l)$, and
- $(l, val_{cl}, val_v) \xrightarrow{act} (l', val'_{cl}, val'_v)$ if $\exists \mathtt{e} = (l, act, G(cl,v), r, l') \in E\, s.t.\, val_{cl}, val_v \models G(cl,v)$, $val'_{cl} = [re \mapsto 0]val_{cl}$, and $val'_{cl}, val'_v \models Inv(l')$,

where for delay $\mathtt{d} \in \mathbb{R}_{\geq 0}$, $val_{cl} + \mathtt{d}$ maps each clock cl in CL to the value $val_{cl} + \mathtt{d}$, and $[re \mapsto 0]val_{cl}$ denotes the clock valuation which maps (resets) each clock in re to 0 and agrees with val_{cl} over $CL \setminus re$.

2.3. Distributed testing of timed systems

In MBT, the model formalizes the set of requirements the SUT is expected to comply with. These requirements are therefore subject to (abstract) test generation (and selection) and checking the conformance relation between the requirements specification model and the SUT, namely, if the exposed i/o behaviour conforms to that of requirements model. A variety of conformance relations exist such as the relation of *testing equivalence*, which relates specification process behaviour to SUT process behaviour in the domain of process algebra [17]. Peleska and Siegel applied this relation to CSP [18]. Springintveld et al. used bi-similarity as a testing relation [19], while Tretmans introduced the input–output conformance (IOCO) relation [20]. The common part of these conformance relations is that they are defined on the model semantics.

The testing methods applied within DTRON rely on the timed version of the IOCO relation. In general, IOCO theory reasons about black-box conformance testing [20]. We say that an implementation I IOCO conforms to a specification S (denoted by $I \sqsubseteq_{IOCO} S$) when at any point in execution it can handle at least as many inputs as the specification and at most as many outputs. The one exception to this rule is that implementation is not allowed to be *quiescent* (i.e. not provide any output) when the specification prescribes at least one possible output [21]. The semantics of the IOCO relation and the related testing theory [20] is originally formulated using LTS and input/output transition systems (IOTS). The conformance relation in timed systems is defined using timed traces of timed IOTS (TIOTS) and of timed LTS (TLTS).

A SUT could be any system that is effectively controllable and observable through test adapters. An adapter is a piece of code that acts as an interpreter between the i/o events of the model and the SUT. An abstract event in the model possibly triggers an executable code in the SUT. This is done by transforming the abstract event to a SUT executable form in the adapter.

Distributed testing presumes handling issues that typically accompany execution in a distributed computational environment. True parallelism and timing imperfections due to hardware/communication jitters are common examples of such issues. Also, timing aspects due to the delays in test adapters need to be accounted for in the IOCO testing of distributed CPS. To minimize the delay effect adapter implementations are ideally kept as simple as possible. If adapters delays start distorting the timing specified in the models, these effects need to be explicitly taken into account also in the test models, and the correctness of tests needs to be re-verified in the presence of these delay effects. Otherwise, although the SUT is a correct wrt model, the extra delay introduced by adapters would violate the test execution timing and give false-negative results.

For time critical systems the correctness of timing needs to be included in the conformance relation, thus, applying the RT-IOCO relation instead of the untimed IOCO was suggested in [22]. Ideally, the RT-IOCO needs to be supported in distributed testing but due to the aforementioned test implementation related issues in distributed systems the RT-IOCO testing is practically infeasible. This has led to the need for relaxing the conformance relation between the model and the SUT. Therefore the notion of a weaker conformance relation – Δ-*testability* [2] – is a main performance consideration the execution environment DTRON is designed for. In brief, the Δ-testability criterion takes advantage of the timing information that is not available in untimed models and accounts for the signal propagation delays of input and output such that wrong input/output interleaving never occurs and the test verdict is correct. In other words, the goal is to minimize the effect of messaging and adapter latency overhead and represent this latency explicitly in the model. While the test model augmented with deployment overhead is still proven to be correct, it avoids producing false test verdicts.

In the case of remote testing, all test inputs are generated by a single centralized tester. This means that the centralized tester will generate an input for a certain SUT port, will wait for the result from some (possibly different) output port, and will continue with the next set of inputs and outputs until the test verdict (passed, failed, inconclusive) can be made. If the SUT is distributed in a way that signal propagation time is non-negligible, this can lead to a situation where the tester is unable to generate the necessary input for the SUT in time specified by the test model. These timing issues can render testing a SUT impossible if the SUT is a distributed system with strict real-time constraints.

To overcome the timing problem, the distributed testing approach described in [23] extends the Δ-testing idea by splitting the monolithic remote tester into multiple local testers directly attached to the ports of the SUT and capable of synchronizing between each other. The adapters still remain between the SUT and the local testers. Thus, instead of bidirectional communication between a remote tester and the SUT, only unidirectional synchronization messages between the local testers are required to update the other testers on the i/o events that occur on a tester's local port. At the same sites as the test ports of SUT the communication delay between a local tester adapter and the SUT port is negligible.

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 the model time), DTRON implements these synchronization channels between local tester models using Spread services, as it will be described in more detail in subsection 3.2. We also assume 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. While in the 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), the distributed testers' communication with local ports can be ignored. Only (unidirectionally) propagating locally observed SUT outputs or locally inserted inputs for synchronization purposes to other testers at different sites are needed, which means the delay for test controllability globally is one Δ.

According to the method introduced in [23], the local testers are generated in two steps: first, a centralized remote tester is generated by applying the reactive planning online-tester synthesis method of [24], and second, a set of synchronizing local testers is derived by cloning the monolithic tester and mapping them to location specific tester instances. The local testers project the global test run to that of observable on local ports of the SUT, and inter-location synchronization events are linked to them. Although the communication overhead of test run increases due to the synchronization traffic, one of the DTRON design solutions, namely multi-cast communication mode, allows compensating for this and reduces the testers' reaction time by half compared to the centralized approach.

3. DISTRIBUTED TEST EXECUTION ENVIRONMENT DTRON

This section presents the DTRON software architecture by providing the details of the subsystems, the communication model, and the integration mechanism. The architecture of DTRON comprises the following subsystems: Uppaal TRON, Spread toolkit, and API.

DTRON [12] extends the functionality of Uppaal TRON [7] by enabling distributed and coordinated execution of tests across networked architectures. The test execution relies on Network Time Protocol (NTP) based on clock corrections to give a global timestamp (t_1) to events arriving at the SUT adapter. These events are then globally serialized and published to other subscribers using the Spread toolkit [25]. Subscribers can be other SUT adapters as well as DTRON instances. Subscribers that have clocks synchronized with the NTP also timestamp the event of receiving a message (t_2) to compute and if necessary and possible, to compensate for the messaging time overhead $\Delta = t_2 - t_1$. The parameter Δ is essential in real-time executions to compensate for messaging delays in test verdicts that may otherwise lead to false-negative non-conformance results for the test runs.

3.1. DTRON architecture

Architecturally, DTRON forms a wrapper around the Uppaal TRON tool by incorporating three other key components:

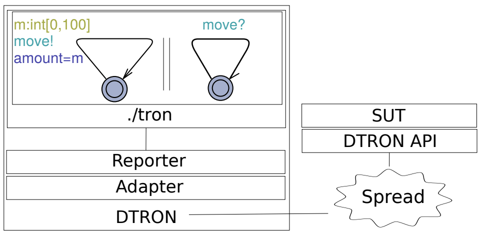


Fig. 2. DTRON configuration for remote testing.

- Reporters/Adapters to communicate directly with instances of the TRON tool and to convert test model symbolic inputs/outputs to data communicated over Spread to the SUT;
- Spread toolkit to interface the runtime with other major languages and platforms and Google Protocol Buffers for message serialization;
- DTRON API that serves as a local test adapter having a direct connection with SUT ports.

The DTRON Adapter is automatically generated by DTRON and linked to the Spread multicast network. The byte-level data traversing in the Spread network is serialized and de-serialized using Google Protocol Buffers. Figure 2 shows a fragment of the DTRON configuration for remote testing. In the following the functionalities of DTRON components and the main design solutions motivated by these key components are described.

3.1.1. Uppaal TRON

Uppaal TRON is a model-based online testing tool based on Uppaal model checking engine. The abstract tests, represented by Uppaal models, are executed and the conformance of SUT to a model is checked simultaneously while maintaining connection to the system in real-time. In order to interface with SUT, an adapter needs to be defined to interpret the model stimuli to SUT and transform the reactions back to I/O symbols of the model. Uppaal TRON provides a C and Java application programming interface (API) for this. The API consists of two classes: the *Reporter* and the *Adapter* (see Fig. 3).

The immediate connection to Uppaal TRON runtime and the underlying UTA model is handled by *Reporter*.

(a)

```
package com.uppaal.tron;
 public interface Adapter {
 void configure(Reporter reporter);
 void perform(int chan, int[] params);
}
```

(b)

```
package com uppaal.tron;
class Reporter extends VirtualThread {
 int addInput (String channel);
 int addOutput (String channel);
 void addVarToInput (int channel, String variable);
 void addVarToOutput (int channel, String variable);
 void setTimeUnit (long microsecs);
 void setTimeout (int timeout_in_units);
 void report (int chan, int[] params)
}
```

Fig. 3. (a) Uppaal TRON *Adapter* class. (b) Uppaal TRON *Reporter* class extract.

Whenever *Reporter* first connects to the runtime, the settings for the following session are configured. This is the 'handshake' phase implemented at method *void configure*(*Reporter*) that is invoked by TRON giving access to the actual *Reporter* object for session configuration. Within the handshake the following are configured:

- the *time unit* that defines how many microseconds the unit of model clock is;
- the length (in microseconds) ofs the timeout that bounds the model execution time;
- inputs–outputs that are declared as channels between the Uppaal model and the SUT.

Note that not all the channels specified in the testing model belong to the test input–output alphabet, but only the ones explicitly declared to be so upon the handshake phase. The other, 'internal', channels are used for synchronization of model processes internally only. By using the API a user can register the *Adapter* to get notified when a synchronization event (denoted by a channel) occurs in the model. An input assignment is declared calling the *int addInput*(*String*) method on the *Reporter* object reference. *String* argument has the value of the corresponding channel name in the model. Outputs are declared respectively by invoking *int addOutput*(*String*). Note that declaring a synchronization as SUT output changes its interpretation in the model. The output channel is not enabled until it has been triggered by *Adapter*. This is done by invoking the *report*(*int*) method on the *Reporter* object reference.

The protocol implementing the immediate connection between the model and *Adapter* is optimized in a way that it does not transport the synchronizations with the actual channel names. When registering inputs and outputs each channel is assigned an integer index instead. This index is returned by the corresponding *addInput* or *addOutput* method and used then to encode which synchronization exactly occurred during the runtime. Keeping track of these indexes is the responsibility of the programmer in TRON, but DTRON hides this complexity behind a more generic object-oriented API.

Whenever a subscribed synchronization event occurs, the *Adapter* method *perform*(*int*, *int*[]) gets called, providing the related information about the event in the argument list. The first integer value denotes the channel index of the synchronization that occurred and the second argument of the integer array gives the integer variable values attached to the event. Attaching variables to a channel is done by the invoking methods *addVarToInput*(*int*; *String*) and *addVarToOutput*(*int*; *String*) on the *Reporter* object reference during handshake. The first integer parameter denotes the *channelId* index bound to have variables attached. The channel needs to be registered first in order to get the index assigned. The second parameter *String* defines the variable name in the model. The second argument of the integer array of the method *perform*(*int*, *int*[]) gives the values of the attached variables. The variable names are not transported, but only their values in the order of declaration when attaching them to a channel. With such an explicit naming mechanism the scalability issues of Uppaal TRON become evident, e.g. the TRON *Adapter* specification gets easily untractable when the models reach the practical 'industrial' size. Making changes to parts of the model that modify the input or output alphabet contract implies immediate changes to the *Adapter* code in TRON as well. This is because the API does not support easy channel/variable mappings. It is left to the responsibility of the developer using the API. This motivated the decision that the API of DTRON needs to abstract away the low-level interfacing details related to channels and provide the API in a domain specific language.

Another problem with the applicability of Uppaal TRON is that it cannot be applied in distributed testing scenarios. This is where the SUT has multiple physical ports, possibly on different machines. This requires a timing-aware messaging medium to coordinate the *Adapters* and enable messaging between multiple Uppaal TRON sessions needed for distributed execution. The primary design objective for introducing an additional messaging layer was not to break the underlying formal semantics of UTA. In the first place, the messages need to preserve their order in distributed execution and, secondly, due to real-time constraints to model execution the messaging transport overhead needs to be kept to a minimum and possibly measurable to enable compensation for sporadically changing communication delays.

3.1.2. Spread toolkit and Google Protocol Buffers

DTRON utilizes the Spread toolkit [25] for implementing the message interchange. The messaging pattern is *publish–subscribe*. Local testers subscribe for messages and get notified by callback methods when new messages are available. The Spread toolkit provides a high performance messaging service that is resilient to faults across local and wide area networks. Spread functions as a unified message bus for distributed applications and provides application-level multicast, group communication and point to point support. Spread services range from reliable messaging to fully ordered messages with virtual synchrony delivery guarantees. DTRON uses Spread API supported *SpreadGroups* to define publishable *SpreadMessages*. Each group has a name and a set of subscribed members. Group members subscribe by joining the group or unsubscribe by leaving it. Spread group messaging relies on a *broker* that is responsible for the actual message queuing mechanism. *Broker* is a server-like program that binds to a network socket to accept requests. Members connect to a broker to subscribe to groups or to publish messages.

Spread has six levels of message delivery guarantees that can be set: *unreliable* messages, *reliable* messages, FIFO (by sender), *casual*, *agreed*, and *safe* (total order). The messages used in DTRON are flagged as *safe* to gain the guarantee they are delivered in exactly the same order as produced to all recipients and not possibly break the underlying UTA semantics. DTRON wraps the Spread API and configures the group memberships and message configuration. DTRON is packaged in two different ways: *embeddable* and *standalone* (executable bundle). During the standalone execution the Uppaal model file given as input is parsed for channel variables prefixed with *i_* (inputs) or *o_* (outputs). The naming convention is given from the adapter perspective – inputs to the adapter (from the SUT) and outputs from the adapter (inputs to model). Whenever an input prefix is found, it is registered with Uppaal TRON API to get notified when it occurred. Also the appropriate *SpreadGroup* is created at the broker or joined if it already exists. Whenever an input channel is then triggered at the model, its name and the associated variable values are published to the appropriate group. Integer variables can also be associated to a publishable channel by prefixing their names with the corresponding channel name. For instance, when having a channel *i_test* and an integer variable *i_test_n*, a message to a group *test* is sent with attached variable *n* with its value. When an output prefix is found in the model, it is registered to be an output in the TRON Adapter API. Whenever a model reaches a state where the outgoing transitions are labelled with output channels, the channels are bound to be disabled until they get triggered by the adapter and become enabled. For each output channel a corresponding Spread group is subscribed to. Whenever a message is then published to this group, the adapter triggers the corresponding labelled transition to be enabled at the model. Similarly to incoming channels, outgoing channels could also have associated integer variables.

The main criteria for choosing the appropriate message serialization framework for DTRON was that it needs to be as fast as possible and support language bindings for at least Java, C/C++, and Python. Since DTRON runtime is expected to operate with around 1 millisecond granularity, it needs a microsecond scale serialization. Google Protocol Buffers [26] meets these requirements and relies on an intermediate data definition language to define message types and structure used for serialization.

3.1.3. DTRON API

DTRON API is mostly for interacting with Spread messaging runtime. DTRON uses this API internally to proxy UTA events via Spread. Users can apply the API also to write adapters to the SUT. Adapters can also implement other functions, e.g. logging, time manipulation, etc. The DTRON API domain model is depicted in Fig. 4. Class *Dtron* is responsible for handling the connection to the Spread broker, i.e. allocation and release of related resources. It serves as a main entry point for the API since this requires a connection. The DTRON connection is used to assign *DtronListeners*.

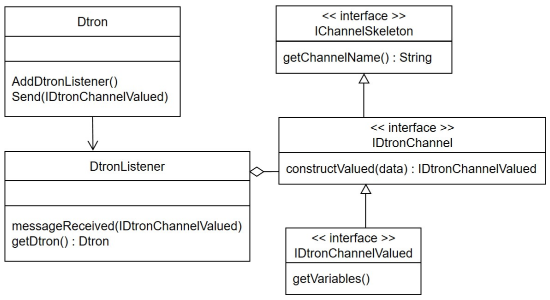


Fig. 4. DTRON API domain model.

Listeners are built based on specific *IDtronChannels* that hold the details about the model channel, i.e. the name and possible variable assignments. Whenever an *IDtronChannel* matching synchronization occurs and is published to a Spread group, the listeners get notified by DTRON invoking the callback method *messageReceived*(*IDtronChannelValued*) and passing the values in the corresponding object as an argument.

To publish events to a Spread group, the class *Dtron* provides a method *send*(). The required argument object cannot be constructed directly. An *IdtronChannel* template has to be constructed first to declare the channel name and related variables. The resulting template object holds the appropriate *constructValued*(*data*) method to assign concrete values to the variables. The assignment is cross-validated to the variable list declared in the template to avoid illegal assignments. While implementing *DtronListener* as anonymous inner classes or a subclass, there is a convenience method *getDtron*() to get a handle to the *send*() method for immediate inline reply back to Spread. The infrastructure code in *DtronListener* constructs a new object based on the provided template with immutable map of variable names and values for intentional use and eliminates direct access to underlying (immutable) pointers. So accidental variable manipulation would not cause the runtime to crash.

3.1.4. Distributed execution

In distributed testing there are multiple physical ports for interactions between the tester and the SUT. A DTRON instance running on one port serves as a local tester and the publish–subscribe messaging allows the observation of a global trace [3]. Figure 5 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 ports for stimuli and observations. Each port used in the test is directly connected to a DTRON instance running against it. This instance can be an *Adapter*, a *Model*, or a combination of both. DTRON instances communicate over Spread, which can be clustered.

The local testers embedded into DTRON instances are interfaced to SUT ports via adapters and subscribed

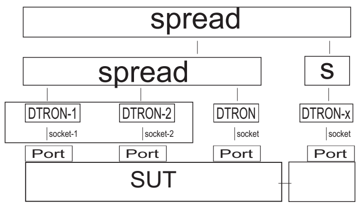


Fig. 5. Distributed testing data-flow in DTRON.

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, which are a mechanism to agree on how time passes and allows for Δ-testability. This addresses the problem of non-uniform message transmission delays between distributed DTRON testers. Consider the Fig. 5 example. 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 $t_2 < t_1$ instead, thus leading 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.

4. PERFORMANCE EVALUATION

The performance evaluation presented in this section aims at estimating the computational impact of a local instance of DTRON upon the latency of test stimuli. The latency upper bound is crucial when deciding on the applicability limits of DTRON to avoid inducing false-negative test verdicts. Scalability and latency are concerned with the number of nodes (and SUT ports) in the network. This directly relates to the performance of Spread since nodes have their local instances of DTRON that run in parallel. So the limiting factor of scalability is rather the Spread toolkit, whose performance depends on the number of communicating nodes and their generated load. According to the analysis in [27 (Fig. 6.14)], the increase of latency between 180 and 350 microseconds of Spread is almost linear up to a throughput of 1500 mbps on a 10-gigabit network. A further increase of the load causes an exponential increase of latency, making further upscaling clearly infeasible. Up to this threshold our measurements have confirmed that, as a rule, the end-to-end latency of two DTRON instances is within the range 0.6–0.7 ms, as will be shown in the following.

Having the load–performance dependence and usability limits of Spread available, the goal of this paper is to determine the latency overhead DTRON has due to introducing an extra layer of messaging abstraction. The focus is on measuring the effect of the Spread toolkit as messaging service with the combination of Google Protocol Buffers. The latency benchmarking is carried out in three different execution environments to demonstrate the scalability with respect to different application constraints. The results are presented with focus on clarifying the DTRON application limits.

4.1. Experiment setup for performance evaluation

The experiment setup is based on the latency analysis model that comes bundled with Uppaal TRON, the *Ticker*. A *Ticker* (see Fig. 6) executes a clock tick every certain time interval. The time interval is designated with variables $p = 250$ and $t = 50$, a guard condition on a reflective transition and a location invariant that forces a tick on average every 250 UTA clock units with the deviation within time interval $t_d = [-50, 50]$. The experiment measures synchronization channel reports (messages) arriving at the TRON Java API as a baseline and then measures the extra delay it takes to pass this information through DTRON. Figure 7 outlines the data flow and timing points. Messages first arrive at the Spread adapter and t_1 times the event. The second timing t_2 is taken when the message arrives at DTRON adapter. The difference $t_\Delta = t_2 - t_1$ is computed and then analysed over the experiment time. We execute this sample model

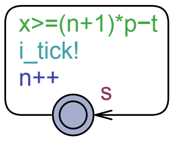


Fig. 6. *Ticker* UTA model.

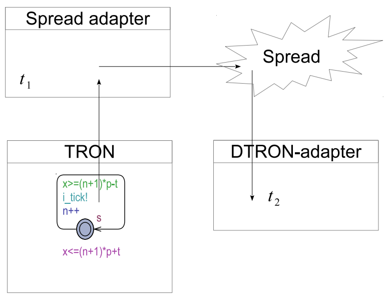


Fig. 7. Experiment setup data flow (architecture).

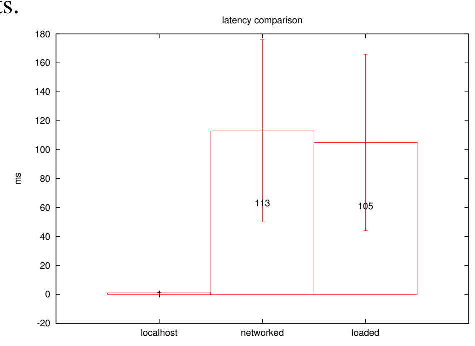
with DTRON only in eager mode and measure the time instances of the arrival of the incoming synchronization event in the TRON Java adapter. ‘Eager’ mode is a feature of Uppaal TRON and is related to the way it handles enabled transitions. In this mode an enabled transition is taken immediately when it becomes enabled.

4.2. Results

Figure 8 shows the results of latency benchmarking experiments in three different execution environments:

1. Messages transmitted over a network loop-back interface. That is, the Spread broker runs on the same machine as the Adapter.
2. Messages transmitted over a switched 1*Gbps* Ethernet network.
3. Messages transmitted over a loaded switched 1*Gbs* network with 50% of the bandwidth allocated by Distributed Internet Traffic Generator [28].

Figure 9 shows empirical results of a Windows operating system networking stack where regardless of some caching symptoms the mean latency drops to a consistent 1 ms. Figure 10 shows aggregated latency results revealing correlation between the latency and computation time. Computation time is emulated to 1 ms by utilizing $sleep(1)$ instruction for simplicity. Instant low computation events overwhelm the networking stack. This in turn causes packet buffers to fill and maintenance subsystems try to compensate. This changes the network throughput and occasional bursts/peaks may occur. Note that *instant computation* (0 ms sleep) is for informational purposes only. Stress test results in Fig. 11 show a mean latency of 5 ms while stimulating events with UTA at maximum capacity. Even with anomalous behaviour exposed as fluctuation at samples 415–420 in Fig. 9 the stimulation computation time is known not to fall below 1 ms by experimental results.



	β	σ	$t\Delta_{\max}$
loop-back	1	0.1	2
network	113	64	221
loaded	105	61	220

Fig. 8. Latency results. β is the mean, σ is the standard deviation, and $t_{\triangle_{\max}}$ is the maximum latency in milliseconds.

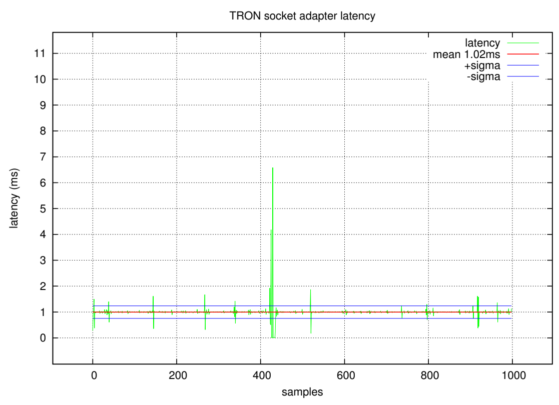


Fig. 9. TRON adapter latency fluctuations, anomaly at samples 415–420.

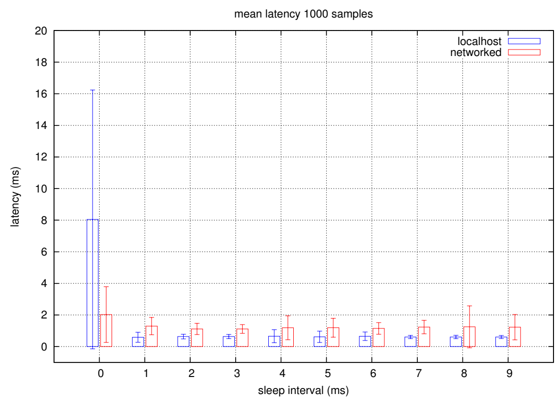


Fig. 10. Aggregated latency results comparing computation time against latency.

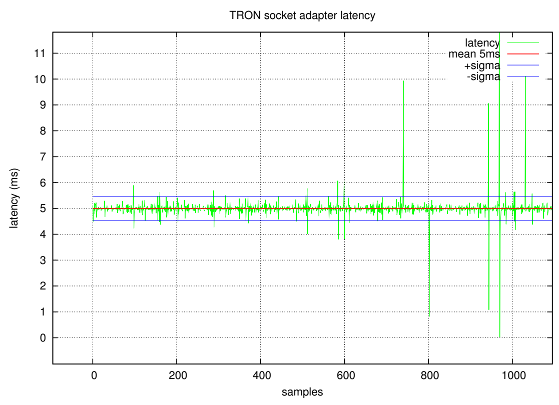


Fig. 11. Eager UTA at 5 ms.

The minimal 1 ms latency lag is caused by the TRON Java adapter that communicates with TRON executing a UTA model over a *SocketAdapter*. The reason is latency due to the networking stack. Given a computationally non-intensive function, the execution can be considered to be instantaneous, which would result in heavy packet-intensive traffic of messages to the Spread network. Since *Nagle's algorithm* for improving the efficiency of TCP/IP networks by reducing the number of packets that need to be sent over the network has been turned off, this would result in a huge overhead of decorating the actual message with TCP/IP packets and in significant loss and fluctuations in the throughput. A sleep interval of 0 ms in Fig. 10 illustrates this scenario. In other words, this shows how the 'networking' aspect affects the real-time performance of DTRON. An 'anomaly' will occur if an adapter does zero work and produces output at maximum speed, essentially 'flooding' the networking stack. This is what the 0 ms sleep interval column in Fig. 10 shows. The performance is still reasonable though. The diagram also shows a deviation of 8 ms that results in a latency of 0 ms. This is the product of the networking stack buffering the packets. On the other hand, when the adapter does at least some work (>=1 ms) one can expect a consistent latency of around 1 ms or less.

Figure 12 shows sleep-vs-latency analysis when using the *nanoSleep*() function for computation simulation. It allows a nanosecond scale control over the thread blocking time instead of milliseconds – that is with regular *sleep*. Although both functions seem to implement the same thing, the *nanoSleep* is internally implemented in a different way, being computationally more intensive. A marginal increase in such computation time results in a substantial drop in latency and its fluctuations.

Since this stress test runs fast, the experiment is carried out with 10 K samples instead of 1 K previously used. This is to demonstrate how it would scale after 1 K. The *nanoSleep()* internal implementation uses processor core ticks to count time instead of a *real-time clock* module. Firstly, this is computationally more expensive than querying a real-time clock. Secondly, although the processor tries to coordinate core times to be equivalent, it is not always guaranteed to be so. At nanoscale resolution it is often the case that processor cores have temporal misalignment that can introduce violation of

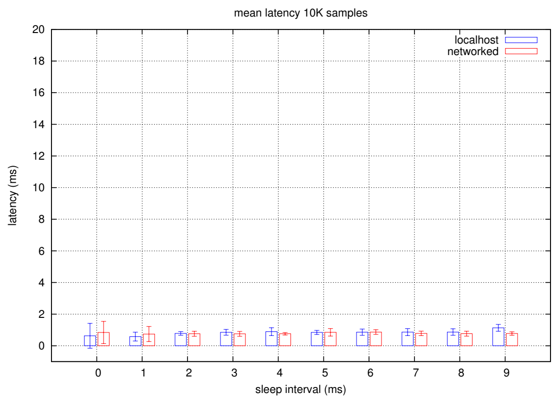


Fig. 12. Aggregated latency using *nanoSleep()*.

hardware clock synchronization. Given the nature of processor coordination of individual tasks distributed around cores, it might happen that the difference between consecutive *nanoTime()* readings turns out to be negative. The clock misalignment does not influence Spread or DTRON. It is about how time is measured on a concrete (hardware) platform during experiments. The *sleep*() function measures time in milliseconds, but DTRON together with Spread is considerably faster than that, hence the need for *nanoSleep()* experiments.

5. USABILITY STUDY

The usability of DTRON functional and performance characteristics outlined in Section 4 has been shown in several distributed testing case studies: a city street lighting control network [29], a ROS-based navigation system [30], an interbank trading and stock exchange system [23], etc.

The city street lighting control network involves up to 1000 controllers spread over the city area (depending on the deployment needs), one central server, and one or more backup servers. The system aims to automate turning city lighting on and off or dimming depending on the atmospheric illumination threshold in different places of the city. Controllers are low-power and low-computation embedded systems. The controller communication medium is General Packet Radio Service (GPRS) over 2G Global System for Mobile communications (GSM). This medium introduces computation and communication delays that are hard to handle with non-distributed MBT methods due to the conformance problems caused by observation time uncertainties. DTRON and Δ-analysis were used to address conformance problems with extensive automated testing.

The SUT was server software. The servers monitor the illumination conditions over the city and, depending on the weather conditions, compute the control settings for local light controllers. Each controller controls a feeder for lights of one or two streets. Controllers regularly initiate communication sessions with the server to update their settings and report on their status. If the connection fails, the controller repeats the session initiation procedure and along failed data also the latest changes are transferred. Controllers work with their earlier settings until they succeed with the new session and settings update. The test suite emulates the sensor data and controller status under different illumination conditions and in the presence of communication failures.

More than 120 test cases were generated by varying controller communication settings (session frequency, duration of the session, number of communication failures per session, location of controller, etc.) and system operator profiles. The model of the user performing actions against the Light controller Web UI was introduced to test simultaneous correction of settings done by operators at different geographic locations (within an area of approximately 30 km diameter). Distributed execution means here that *n* instances of the operator model were executed at different sites to simulate load and possible 'state' faults that could occur on the server side due to their possible interference. Although the timing constraints in this case study were not critical, generating the short distributed load burst patterns of over 1000 controllers presumed server side message receiver modelling in 10 ms scale. This load did not cause false conformance violations (due to DTRON performance limits) between the SUT and the model.

The case study of ROS packages testing addressed in [30] involves high level robot control, such as e.g. localization and navigation of mobile robots. The approach of Robot Unit Testing by Bihlmaier and Wörn [31] is extended in two ways: first by introducing a white box metric of code coverage, in particular statement and branch coverage, and second, by combining a DTRON based testing into the test set-up, which allows formalizing the requirements of the stack of ROS packages together with either a real or simulated set of sensors and actuators. The experiments involve modelling and testing the navigation and localization components of the software stack developed in the STRANDS project [32]. The stack was chosen as the SUT because it involves multiple layers of functionality on top of the standard ROS *move_base* mobile base package responsible for accomplishing navigation. It is open sourced, accessible on GitHub, contains a working simulation environment built using Morse [33], and many existing quality assurance techniques are actively used in the project, including unit tests and a Jenkins based continuous integration system. The test goal is to check if the robot maintains correct (in the sense of navigation constraints) behaviour in the presence of multiple static and moving obstacles (humans/other robots). The appearance of static obstacles (30 objects) and dynamic obstacles (5 and 9 objects) was the test input to the SUT whereas the inputs were distributed in the virtual environment of a building floor. Robot reactions were the test outputs. The delays of receiving/sending respectively inputs/outputs were due to the execution time of the software stack. The test suite involved different scenarios, i.e. high level test cases. The tests were run in two different simulated environments and repeated in the same scenarios by sending the goals to the *move_base* and *topological_navigation* action servers. Altogether 12 high level test scenarios were generated and executed in two different virtual environments. In the best case the topological navigation test provided code coverage respectively by packages of the ROS stack: action_lib – 54%, strands navigation – 17%, topological navigation – 28%, localization node – 73%, and navigation node – 73%. In the presence of none dynamic objects DTRON could perform fast enough to maintain the realistic repositioning of those objects and processing robot reactions when visiting the navigation waypoints at a speed of 1.4 m/s.

The third case study [23] includes a fragment of an interbank trading system (ITF). Four ports of the SUT are involved in the test configuration representing respectively clients and banks that are geographically located in different places. In such a situation, the propagation of the input and output signals is not negligible and may affect the interbank bidding processes. Each port consists of inputs and outputs, but not necessarily both. The ports represent the quotes (bid and ask prices), order requests, and order confirmations. The test configuration consists of two banks (A, B), interbank market (M), and a client (C). The client (C) is connected to each bank information system. The banks are connected to the clients and to the interbank market (M). The test suite includes four cases where a client wants to engage in arbitrage and therefore is waiting for a situation where the banks have different prices for the same financial instrument. The test scenario covers the situations where the client receives the prices from banks and sends an order request (buy or sell) to the bank they want to buy or sell that financial instrument. The banks receive the interbank prices from the interbank market and they buy and sell the instrument they do not possess themselves from that market for order clearing. An arbitrage opportunity arises when one bank's information system has different prices than the price at the market. The bank forwards the price that it perceives to be the current price to the client, but in reality the price has already changed in the market. This issue is completely due to latency, and testing the performance of the ITF requires distributed testers instead of one centralized remote tester to be as close as possible to real low latency bidding situations.

In all these cases the integration testing time was reduced on average by 24% due to the automated generation of tests and easier test deployment and execution. The most time consuming parts of the testing process were formalization of requirements and test case definition.

The DTRON testing experiments were conducted by authors of references [29] and [23] in parallel with the development team who scripted tests in TTCN-3 manually and ran them on the commercial tool TestCast (http://www.elvior.com/). The test suites compared were almost identical in their number of test cases and coverage. The test development effort in time was estimated in both cases and compared. When the commercial development team tried modelling the test cases in UML state charts and rendering them into TTCN-3, it took longer than direct scripting in TTCN due to the learning curve of the proper usage of UML. The test execution time itself was negligible compared to the test definition and modelling time. The advantage of using UTA was revealed also in modelling timing constraints on more abstract level than that of TTCN-3 language where timeouts need to be given explicitly in the code for each expected test reaction. The UTA model used in the case study in [30] was generated completely automatically based on the robot navigation map, navigation goal (waypoints), and obstacles that influence the trajectory planning. The same test cases were programmed also manually and their coverage was compared. Together with creating the test adapters and test deployment the total test development and test execution time using models and DTRON was almost 35% shorter compared with the testing process where the tests were written manually in C++.

The testing of the city lighting project revealed a timing non-conformance bug when executing multiple copies of testers mimicking different users where one or the other user was denied access to the system. The erratic occurrence of this bug could not be made reproducible, but it gave enough insight to the developer to fix the (session management) problem.

The bug described in [29] was detected when the system operator models were executed simultaneously at *n* different sites to hit the server. There occurred a server-side session management failure where some of the operators (model execution instances) were denied access to the system and became forcibly logged out. The SUT exhibited the expected behaviour in most of the cases, but for some cases the server showed the lights to be off when the controllers had actually the lights on and the status was expected to be synchronized with the server. An example test run was screencaptured and is available for design analysis on DTRON website [12].

Similarly, the ROS case study published in [30] revealed an incorrect mapping of virtual coordinates of the waypoints in the laser scan map. This appeared to be a mistake in the navigation task specification rather than a bug in the ROS stack.

6. CONCLUSION

This paper presents the architectural design solutions and the results of computational (and timing) overhead analysis of DTRON, a distributed test execution tool. DTRON, as one of the first model-based test execution environments for distributed testing of systems with strict timing constraints, is specifically designed to reduce the runtime overhead and minimize the tester reaction time to make it suitable for low response time Δ-testing. The resulting framework, capable of operating in the millisecond scale, is of acceptable precision for a wide spectrum of cyber-physical systems. Three case studies highlighted in the paper assure that DTRON has practical value in various testing contexts ranging from robot real-time navigation to geographically distributed systems testing. Future work will include the implementation of monitors within DTRON for collecting online data about the network non-static latency characteristics that may violate preset Δ-parameters of remote and distributed tests.

ACKNOWLEDGEMENTS

This research was supported by Tallinn University of Technology. The publication costs for this article were covered by the Estonian Academy of Sciences and Tallinn University of Technology.

REFERENCES

1. Lee, E. A. Cyber-Physical Systems – Are Computing Foundations Adequate? Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap, October 16–17, 2006, Austin, TX. NSF, 2006.
2. David, A., Larsen, K. G., Mikučionis, M., Nguena Timo, O. L., and Rollet, A. Remote testing of timed specifications. In *Testing Software and Systems: 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13–15, 2013, Proceedings* (Yenigün, H., Yilmaz, C., and Ulrich, A., eds). Lecture Notes in Computer Science, Vol. 8254. Springer, Berlin, 2013, 65–81.
3. Hierons, R. M. Oracles for distributed testing. *IEEE Trans. Softw. Eng.*, 2012, **38**(3), 629–641.
4. Hierons, R. M., Merayo, M. G., and Núñez, M. Using time to add order to distributed testing. In *Proceedings of FM 2012: Formal Methods: 18th International Symposium* (Giannakopoulou, D. and Méry, D., eds). Lecture Notes in Computer Science, Vol. 7436. Springer, 2012, 232–246.
5. Ponce-de Léon, H., Haar, S., and Longuet, D. Distributed testing of concurrent systems: vector clocks to the rescue. In *Theoretical Aspects of Computing – ICTAC 2014: 11th International Colloquium, Bucharest, Romania, September 17–19, 2014. Proceedings* (Ciobanu, G. and Méry, D., eds). Lecture Notes in Computer Science, Vol. 8687. Springer International Publishing, 2014, 369–387.
6. Khoumsi, A. Testing distributed real time systems using a distributed test architecture. In *Proceedings of Sixth IEEE Symposium on Computers and Communications*. IEEE, 2001, 648–654.
7. Uppaal TRON. `http://people.cs.aau.dk/~marius/tron/` (accessed 2016-11-24).
8. Krichen, M. and Tripakis, S. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 2009, **34**(3), 238–304.
9. Ru Dai, Z., Grabowski, J., and Neukirchen, H. TimedTTCN-3 – a real-time extension for TTCN-3. In *Proceedings of Testing of Communicating Systems XIV – TestCom 2002*. IFIP – International Federation for Information Processing (Schieferdecker, I., König, H., and Wolisz, A., eds), Vol. 82. Springer US, 2002, 407–424.
10. TestComplete. Date of last access: 18/11/2016, `https://smartbear.com/product/testcomplete/overview/` (accessed 2016-11-24).
11. Öztaş, G. Testing distributed real-time systems with a distributed test approach. Master's thesis, The Graduate School of Natural and Applied Sciences of Middle East Technical University, 2008. `http://citeseerx.ist.psu.edu/viewdoc/download?`

doi=10.1.1.632.5375\&rep=rep1\&type=pdf (accessed 2016-11-24).

12. DTRON. https://cs.ttu.ee/dtron/ (accessed 2016-11-18).
13. Larsen, K. G., Pettersson, P., and Yi, W. UPPAAL in a nutshell. *Int. J. Softw. Tools Technol. Transf.*, 1997, **1**, 134–152.
14. Model-based design. https://en.wikipedia.org/wiki/Model-based_design (accessed 2016-11-24).
15. Anier, A. and Vain, J. Model based continual planning and control framework for assistive robots. In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems, Rome, Italy* (Benavente-Peces, C., Ali, F., and Filipe, J., eds). SciTePress, 2012, 403–406.
16. Behrmann, G., David, A., and Larsen, K. G. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems* (Bernardo, M. and Corradini, F., eds). Lecture Notes in Computer Science, Vol. 3185. Springer, Berlin, 2004, 200–236.
17. Nicola, R. D. and Hennessy, M. Testing equivalences for processes. *Theor. Comput. Sci.*, 1984, **34**(1–2), 83–133.
18. Peleska, J. and Siegel, M. Test automation of safety-critical reactive systems. *South Afr. Comput. J.*, 1997, **19**, 53–77.
19. Springintveld, J., Vaandrager, F., and D'Argenio, P. R. Testing timed automata. *Theor. Comput. Sci.*, 2001, **254**, 225–257.
20. Tretmans, J. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*. North-Holland Publishing Co., Amsterdam, The Netherlands, 1994, 257–276.
21. Timmer, M., Brinksma, E., and Stoelinga, M. Model-based testing. In *Software and Systems Safety – Specification and Verification*, 2011, 1–32.
22. Hessel, A., Larsen, K. G., Mikučionis, M., Nielsen, B., Pettersson, P., and Skou, A. Testing real-time systems using UPPAAL. In *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers* (Hierons, R. M., Bowen, J. P., and Harman, M., eds). Springer, Berlin, 2008, 77–117.
23. Vain, J., Halling, E., Kanter, G., Anier, A., and Pal, D. Model-based testing of real-time distributed systems. In *Proceedings of 12th International Baltic Conference on Databases and Information Systems*. Springer, 2016, 1–14.
24. Vain, J., Raiend, K., Kull, A., and Ernits, J. P. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE'07*. ACM, New York, NY, USA, 2007, 363–372.
25. The Spread toolkit. http://spread.org/ (accessed 2016-11-24).
26. Protocol buffers. Google's Data Interchange Format, 2011. https://developers.google.com/protocol-buffers/ (accessed 2016-11-8).
27. Babay, A. The accelerated ring protocol: ordered multicast for modern data centers. Master's thesis. The Johns Hopkins University, 2014. http://www.dsn.jhu.edu/~yairamir/Amy_MSE_thesis.pdf (accessed 2016-11-26).
28. Avallone, S., Guadagno, S., Emma, D., Pescapè, A., and Ventre, G. DITG distributed internet traffic generator. In *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference*. IEEE Computer Society Press, 2004, 316–317.
29. Vain, J., Anier, A., and Halling, E. Provably correct test development for timed systems. In *Databases and Information Systems VIII: Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014*, Vol. 270. IOS Press, 2014, 289–302.
30. Ernits, J., Halling, E., Kanter, G., and Vain, J. Model-based integration testing of ROS packages: a mobile robot case study. In *2015 IEEE European Conference on Mobile Robots*. IEEE, 2015, 1–7.
31. Bihlmaier, A. and Wörn, H. Robot unit testing. In *Proceedings: Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20–23* (Brugali, D., Broenink, J. F., Kroeger, T., and MacDonald, B. A., eds). Springer International Publishing, 2014, 255–266.
32. FP7 EU Project: STRANDS. Spatio-Temporal Representation and Activities for Cognitive Control in Long-Term Scenarios. http://strands.acin.tuwien.ac.at/ (accessed 2016-11-25).
33. Echeverria, G., Lemaignan, S., Degroote, A., Lacroix, S., Karg, M., Koch, P., et al. Simulating complex robotic scenarios with morse. In *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR'12*. Springer-Verlag, Berlin, 2012, 197–208.

Ajakriitiliste rakenduste mudelipõhise hajustestimise vahend DTRON

Aivo Anier, Jüri Vain ja Leonidas Tsiopoulos

Küberfüüsikalised süsteemid (KFS) pakuvad suuri võimalusi, kuid ka suuri väljakutseid mitmes valdkonnas, näiteks elektroonikatööstus, transpordisüsteemid ja tööstuse automatiseerimine. Väga keeruka KFS-i tarkvara disaini korrektsuse tagamine nõuab uusi arendusmetoodikaid ja vahendeid, mis peavad olema suunatud laiale arhitektuurilahenduste spektrile. Samuti peavad KFS-i arendusvahendid lahendama olulise paralleelsuse ja ajastamiskitsendustega seotud probleeme. Käesolevas artiklis on käsitletud mudelipõhise testimise vahendit DTRON, mis on välja töötatud ajatundlike hajusarhitektuuriga süsteemide testimiseks. DTRON on loodud mudelkontrollivahendi Uppaal ja *online*'i testimisvahendi TRON baasil, laiendades nende funktsionaalsust *online*'i hajustestimiseks vajalike koordineerimis- ning sünkroniseerimisfunktsioonidega. Hajustestide juhitavuse tagamiseks on DTRON-i projekteerimisel lähtutud Δ-testitavuse nõudest. Artiklis on esitatud DTRON-i arhitektuurilahendus ja analüüsitud selle jõudlusnäitajaid, arvesse võttes võrguühenduse ning testiadapteritest tingitud hilistumisi. Jõudluseksperimentide abil on näidatud, et implementeerimiseks kasutatud vahevara Spread sõnumite järjestamisteenus ja võrgu ajakorraldusprotokoll Network Time Protocol võimaldavad kahandada hajustestide juhitavuse tagamiseks vajaliku parameetri Δ alla 1 ms piiri. See näitaja on piisav paljude võrkarhitektuuriga küberfüüsikaliste süsteemide hajustestimiseks. DTRON-i rakendatavust valideerivad kolm rakendusnäidet: tänavavalgustussüsteemi kontrollerite võrgustiku, pankadevahelise kauplemissüsteemi ja mobiilse roboti navigatsioonisüsteemi testimine.