Quantified Dynamic Metric Temporal Logic for Dynamic Networks of Stochastic Hybrid Automata

Alexandre David*, Kim G. Larsen*, Axel Legay†, Guangyuan Li‡ and Danny Bøgsted Poulsen*
* Department of Computer Science, Aalborg University
† INRIA, Rennes
‡ Chinese Academy of Science

Abstract—**Multiprocessing systems are capable of running multiple processes concurrently. By now such systems have established themselves as the defacto standard for operating systems. At the core of an operating system is the ability to execute programs and as such there must be a primitive for instantiating new processes - also programs are allowed to die/terminate. Operating systems may allow the executing programs to split (spawn) into more computational threads in order to let programs take advantage of concurrent execution as well. One of the most used modelling languages, Timed Automata, is based on multiple automata interacting thus they easily model the concurrent execution of programs. However, this language assumes a fixed size system in the sense that automata cannot be created at will but must be instantiated when the overall system is created. This is in contrast with the fact that developers are able to create threads when needed. In this paper we present our continued work to incorporate spawning of threads into UPPAAL SMC. Our modelling language, *Dynamic Networks of Stochastic Hybrid Automata*, is essentially Timed Automata extended with a spawning primitive and a tear-down primitive. The dynamic creation of threads has the side-effect that it is no longer possible to use ordinary logics to specify behaviours of individual threads - because the threads no longer have unique names. In this paper we propose an extension of *Metric Temporal Logic* with means for quantifying over the dynamically created threads. This makes it possible to zoom in on individual threads and specify requirements to their future behaviour. Furthermore, we present a monitoring procedure for the logic based on rewriting formulas. The presented modelling language and the specification language have been implemented in UPPAAL SMC version 4.1.18.**

I. INTRODUCTION

Computer systems of today are beyond the state where they were statically encoded entities that were disconnected from their surroundings. Instead many software architectures are build with communication to other systems in mind thus each system may rely on other systems for accomplishing their tasks. The systems providing services to other systems must incorporate concurrency into their execution platforms in order to handle requests from multiple clients simultaneously - and to support an unknown number of clients they must be able to make new computational threads at will. Luckily, most main-stream programming languages and operating systems support concurrency out of the box and alleviate the programmer from the burden of programming the concurrency model.

Reasoning on dynamic systems poses a major challenge to the formal methods community, that is the one of being able to develop models and techniques for systems whose state-space is not known a priori. Additionally, it also requires to deal with communication between processes at run time. Process algebras, e.g. CSP [14] and CCS [17] have been designed to analyse such systems. In process algebras the behaviour of systems is described with a minimal set of primitives and they allow us to reason about the equivalence of systems using bisimulation relations. By adding a recursion/replication operator we can express spawning of processes. Whereas process algebras have been developed for reasoning about dynamic systems, we note that few formal tools support dynamic creation of processes. Instead, they require specifying all processes in advance which forces the modeller to encode an underlying resource manager with a preset finite capacity. This stands in deep contrast to the support given by operating systems and programming languages.

In a recent work [9], we developed a modelling framework that allowed spawning inside UPPAAL SMC [8]. In our setting, processes are spawned from a finite set of templates, each being an arbitrarily complex input/output timed system. In this setting, processes communicate via an input/output mechanism of actions, and each of those actions may eventually lead to the creation of one or several new processes. The model checking problem is known to be undecidable for such systems. As a solution, we proposed to equip our system with a stochastic semantic, allowing us to unleash the power of simulation-based solutions such as Statistical Model Checking (SMC)[22]. In this framework, verification reduces to monitor several executions of the system and then use an algorithm from statistics to deduce the overall correctness with a controllable confidence. One of the drawbacks of the work in [9] is that the logic for specifying properties of template systems was a limited extension of Metric Temporal Logic[16] (MTL), where quantifications over processes were allowed at the atomic level. While such a logic is powerful enough to express properties like: "at any time, all the process shall avoid state x", it does not allow to deal with more complex requirements such as "if there exists a process that reaches state q, then it should reach state q' in less than ten units of time after reaching q".

In this paper, we propose an extension of MTL for dynamic timed systems, where quantifications over process templates

Work partially supported by NSFC(61361136002) for IDEA4CPS and the 863 Hi-tech Reseach and Development Program of China(2012AA011206).

can be nested. Our new dynamic quantitative MTL logic is inspired by those proposed to specify properties of infinite state systems, especially in the context of the so-called regular model checking approach [3, 1]. As a second contribution, we present a monitoring procedure for this new logic. The procedure uses rewriting techniques of subformulas of the logic. Our work has been implemented in UPPAAL SMC, the SMC extension of UPPAAL.

Related Work.: Dynamic creation of processes is already part of extensions of process algebras. An example is the fork calculus [12] that extends CCS with a fork primitive. These extensions do not consider quantities and run time verification of complex requirements expressed in MTL. Recently, Sharifloo proposed to avoid this assumption by combining verification and run-time of the deployed system within the Lover framework [19]. This work is in line with our objective, but ignores timed and stochastic aspects. Tools such as BIP have been extended to deal with dynamical architecture [5]. BIP focuses on interactions, while UPPAAL SMC proposes a quantitative framework. Other approaches such as PRS also consider dynamical networks. However, they remain at a highly theoretical level, mostly studying what is decidable and what is not [20]. Those approaches do not consider effective and efficient algorithms. Finally, Henzinger et al., have also considered dynamical extension of *reactive modules* with an application to systems biology. The theory presented in [11] is without a run-time monitoring procedure and the verification process is limited to conformance. There are also a wide range of dynamical architectures dedicated to a specific problem [10]. Our approach is more generic and hence incomparable to those approaches. Dynamic process creation is already part of the model checking tool SPIN[15]. Contrary to our work, SPIN does not consider *timed*, *hybrid* or *stochastic* aspects of systems. Boudjadar et al. [4] have developed a framework called Callable Timed Automata (CTA) that allows dynamic creation of processes. Our work distinguishes itself from theirs by having a stochastic semantics and their work did not consider a logic for expressing properties of the dynamic systems.

II. CLIENT-SERVER EXAMPLE

The purpose of this section is to provide the intuition behind our modelling formalism by means of an example. The example we consider is that of a client-server system shown in Fig. 1. Due to *Traffic* we have clients arriving in the network (1). These clients connect to a server (2) that generates threads (3) to handle the subsequent communication with the clients (4). When the exchange is over, the client and corresponding server threads terminate. This models the typical behaviour of servers that *listen* on a port, *accept* a connection, and delegate the connection to a forked process or a new thread while returning to listening on its port. We aim towards only giving the intuition behind our formalism in this section and leave the actual semantics for later. In short, a model consists of *templates* that we can instantiate to running processes. A process is either instantiated as an initialisation step of the model, or by being spawned by a running process [1].

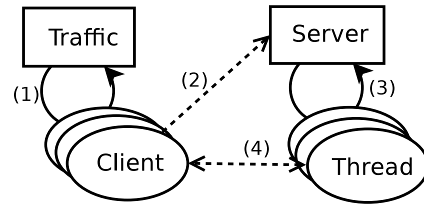

Figure 1: A client-server example.

The model has four templates shown in Fig. 2 through 5.

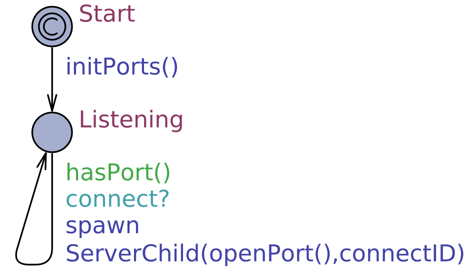

Figure 2: Server()

a) Server.: Figure 2 shows the template modelling the server. To model that servers can accept only a limited number of connections, the server manages its available connection ports with an array. The first transition from Start initialises this array with initPorts(). The server awaits a connection request from a client with a channel synchronisation on connect?. The server reacts only if it has some available port (condition hasPort()) and spawns a server child. This is done with the spawn command that instantiates the ServerChild template (modelling a thread) with argument values computed on-the-fly. The server allocates a port with the function openPort() and forwards connectID that it receives from the client to the server child.

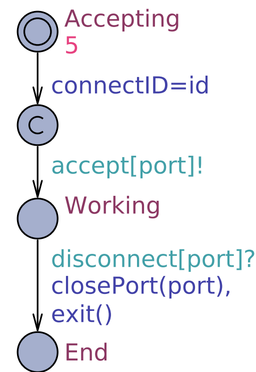

Figure 3: ServerChild(**const** pid_t port, **const** int id)

[1] In the same manner as you spawn processes in computer systems

b) ServerChild.: The template taking care of the connections is shown in Fig 3. An instance of this template starts by taking some time to reply and accept the connection. The time is picked with an exponential distribution with rate 5. Then the instance synchronises back with the client that initiated the connection and "sends" the allocated port number with the synchronisation accept[port]![2]. Here connectID is used to filter out the right client. The location Working abstracts from the actual communication until the client closes it, which is done by the synchronisation disconnect[port]?. The server child closes the port (makes it available again) and terminates, which is done with the special function exit().

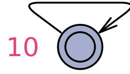

Figure 4: Traffic ()

c) Traffic.: To model traffic, the template of Fig. 4 generates clients, i.e., *spawns* client processes with the expression spawn Client(++clientID). The time between creation is picked with an exponential distribution with rate 10. Each new client receives a unique identifier.

d) Client.: When spawned, the client of Fig. 5 will take some time to connect (exponential distribution with rate 10). It will then wait for the synchronisation accept[p]? that passes the port. The client tests if the reply matches its ID with id==connectID. This is needed since we abstract from the actual communication protocol. The client times-out after 5 time units and will retry MAX_RETRIES times before aborting. If the connection is accepted then the client works for some time, then disconnects with disconnect[port]! and terminates. Here again the client process terminates by calling exit().

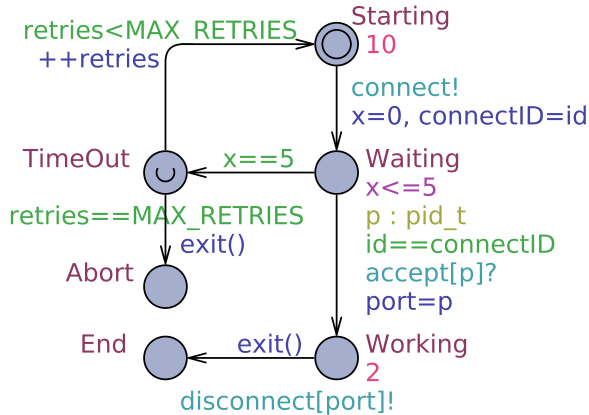

Figure 5: Client(**const** int id)

[2]The trick uses an array of channels for message passing.

e) Support for Dynamic Processes.: When the special command spawn is encountered, UPPAAL SMC creates a new instance of a given template with the current values of the expressions used for arguments. When the special function exit() is executed in a dynamic process, UPPAAL SMC discards this process. The templates that can be instantiated dynamically are declared to be *dynamic* in the global declaration of the model as shown in Listing 1. We also show the functions for opening and closing ports.

```
pid_t openPort() {
  assert(freePort>=0); return ports[−−freePort];
}
void closePort(pid_t p) {
  assert(freePort<MAX_CONNECTIONS);
  ports[freePort++] = p;
}

dynamic ServerChild(const pid_t port, const int id);
dynamic Client(const int id);
```

Listing 1: Global declarations of the client-server example

III. Dynamic Network of Hybrid Systems

In this section we provide the semantical part of our modelling framework. The framework is equivalent to the one we presented in [9] but the semantics are expressed differently: in our previous work all processes were anonymous whereas we in this work give them identities by giving them a name. We later use these names in defining the semantics of our logic.

We abstract from the actual modelling formalism and define our semantics on the basis of timed IO-transition systems (TIOTS).

Definition 1 (Timed IO-transition System): A Timed IO-transition system over the input actions Σ_i and output actions Σ_o is a tuple $(S, s_0, \rightarrow)$ where

- S is a set of states, $s_0 \in S$ is the initial state and
- $\rightarrow \subseteq S \times (\Sigma_i \cup \Sigma_o \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation.

Let $(S, s_0, \rightarrow)$ be a TIOTS then we write $s \xrightarrow{a!} s'$ whenever $(s, a, s') \in \rightarrow$ and $a \in \Sigma_i$. Similarly we write $s \xrightarrow{a?} s'$ if $a \in \Sigma_o$ and $s \xrightarrow{d} s'$ if $d \in \mathbb{R}_{\geq 0}$. In accordance with the compositional specification for timed systems [7] we assume any TIOTS is input-enabled i.e. for any state s and any input action a there exists s' s.t. $s \xrightarrow{a?} s'$. Also we assume determinism thus if $s \xrightarrow{x} s'$ and $s \xrightarrow{x} s''$ then $s' = s''$. Since we assume determinism we denote the $x-$ successor, $x \in (\mathbb{R}_{\geq 0} \cup \Sigma_i \cup \Sigma_o)$ of s by s^x i.e. $s \xrightarrow{x} s^x$.

Timed IO-transition systems can be generated by various formalisms. Well-known formalisms include Timed Automata [2] and Hybrid Automata [13], where states have the form (ℓ, v) where ℓ is the current control location of the automaton and v is a valuation that assigns values to its continuous variables e.g. clocks, costs and hybrid variables. A *discrete* transition from (ℓ, v) to (ℓ', v') corresponds to an edge, in the automaton, between ℓ and ℓ' whose guard is enabled by v. The resulting v' is obtained by performing the updates required by the edge. In *delay* transitions, the values of the continuous

variables are changed according to a flow function that gives the rate of change for each clock. For a timed automaton this rate is always 1 hence a delay of d would increase all variables by d. For hybrid systems the rates are specified using differential equations.

Example 1: Consider the model of a client attempting to establish a connection to a server shown in Fig. 5. This timed automaton has one clock `x` with a starting value of 0. The client is initially in the location `Starting`. From this initial state a possible transition sequence is:

$$(\texttt{Starting}, \texttt{x} = 0) \xrightarrow{0.8} (\texttt{Starting}, \texttt{x} = 0.8) \xrightarrow{\texttt{connect!}} (\texttt{Waiting}, \texttt{x} = 0)$$

In our framework a system consists of a dynamically evolving set of processes, where processes are running instances of templates and can spawn other processes. The available set of templates that a process can be spawned as is given in terms of a *Template Collection* $\mathcal{J} = (\mathcal{T}_1, \ldots, \mathcal{T}_n)$ where each template defines a TIOTS. All the templates share a common set of actions, Σ. This set is partitioned into disjoint subsets $\Sigma^1, \Sigma^2, \ldots \Sigma^n$ and template $\mathcal{T}_i$ uses Σ^i as output actions and $\Sigma \setminus \Sigma^i$ as input actions.

Definition 2 (Template Collection): A Template Collection over the set of actions Σ partitioned into n disjoint sets $\Sigma^1, \Sigma^2, \ldots, \Sigma^n$ is a tuple $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$ where for all i, $\mathcal{T}_i = (S^i, s_0^i, \rightarrow^i, \Psi^i)$ with:

- $(S^i, s_0^i, \rightarrow^i)$ is a TIOTS with output action Σ^i and input actions $\Sigma \setminus \Sigma^i$,
- $\Psi^i : S^i \times \Sigma^i \rightarrow 2^{\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n\}}$ describes for each state-action-pair a set of templates that should be spawned while doing that action from that state.

As mentioned earlier the semantics is based on a dynamically evolving set of processes. When spawning a process, on the basis of a template $\mathcal{T}$, a name p for the new process is extracted from an infinite but countable set of names PNames, and the global state is updated to point p to the initial state of $\mathcal{T}$. Furthermore, in the global state we record that p has type $\mathcal{T}$ and record that the name p has been used.

Formally, a state of a template collection $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$ has the form (Active, T, f), where

- $\mathsf{Active} \subseteq \mathsf{PNames}$ contains the names that has been bound to form a process,
- $T : \mathsf{Active} \rightarrow \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ gives the template type of each process and
- $f : \mathsf{Active} \rightarrow \bigcup_{i=1}^{n} S^i$ maps the bound names to their corresponding state.

Naturally, we require *consistency* in the sense that if $f(p)$ points to a state of $\mathcal{T}_i$ then $T(p) = \mathcal{T}_i$. For the actual spawning of processes we define an operator $\oplus$ defined between a state and a template.

$$(\mathsf{Active}, T, f) \oplus \mathcal{T} = (\mathsf{Active} \cup \{p\}, T', f'),$$

where $p \in \mathsf{PNames} \setminus \mathsf{Active}$, $f'(p) = s_0$, $T'(p) = \mathcal{T}$ and for all $p' \in \mathsf{Active}$, $f'(p') = f(p')$ and $T'(p') = T(p')$. This operator is straightforwardly generalised to sets of templates.

Remark 1: In the above the names for processes was chosen non-deterministically. To make this selection deterministic we will assume an ordering on PNames and always extract the smallest element from Active. Also, we will assume an ordering among templates so that when spawning a collection of templates they are spawned in a deterministic order.

For simplicity we let each template be initially be instantiated by one process thus the initial state is defined as $(\emptyset, _, _) \oplus \{\mathcal{T}_1 \ldots, \mathcal{T}_n\}$. This can easily be modified to only spawn a subset of the processes. The transition relation of a template collection is defined in Fig. 6. The semantics states that the entire system can delay if all processes can participate in the delay. Regarding actions, the system can perform an action $a!$ if there exists some process, of the template $\mathcal{T}$ owning the action, that can perform it and all other components change state in accordance with the input. The processes of $\mathcal{T}$ not performing the action simply ignore it.

DELAY $(\mathsf{Active}, T, f) \xrightarrow{d} (\mathsf{Active}, T, f')$
if $d \in \mathbb{R}_{\geq 0}$ and for all $p \in \mathsf{Active}$, $f(p) \xrightarrow{d} f'(p)$;

ACTION $(\mathsf{Active}, T, f) \xrightarrow{a!} (\mathsf{Active}, T, f') \oplus P$
if $a \in \Sigma^j$, and there exists $p \in \mathsf{Active}$ such that:
$T(p) = \mathcal{T}_j$, $f(p) \xrightarrow{a!} f'(p)$, $P = \Psi^j(f(p), a))$,
and for all $q \in (\mathsf{Active} \setminus \{p\})$, if $T(q) = \mathcal{T}_j$
then $f(q) = f'(q)$, otherwise $f(q) \xrightarrow{a?} f'(q)$.

Figure 6: Transition relation of a Template collection

Example 2: In Fig. 7 we show a graphical representation of the location changes of the clients. In the plot we see that all the clients are spawned in location `Starting` and after some delay move into `Waiting`. Then after some time has passed they are accepted and enter the `Working` location. From there they disappear when entering the `End` location

A timed run of a template collection $\mathcal{J}$ is an infinite sequence $\omega = \mathbf{s}_0 d_0 \mathbf{s}_1 d_1 \ldots$ such that $\mathbf{s}_0$ is the initial state of $\mathcal{J}$, and for all $i \in \mathbb{N}$: $d_i \in \mathbb{R}_{\geq 0}$ and $\mathbf{s}_i \xrightarrow{d_i} \xrightarrow{a_i} \mathbf{s}_{i+1}$ for some $a_i \in \Sigma$. An infinite run is called time-diverging if for any constant $c \in \mathbb{R}_{\geq 0}$ there exists j such that $\sum_{i=0}^{j}(d_i) > c$. For the remainder of this paper we will require all runs to be diverging and for a template collection $\mathcal{J}$ we will let $\Omega(\mathcal{J})$ be the set of all such runs.

Consider each template $\mathcal{T}$ has a finite set of atomic propositions $AP_{\mathcal{T}}$ that can be true in states of that template. Now let s be a state of $\mathcal{T}$ then $P^{AP_{\mathcal{T}}}(s)$ gives the finite subset of $AP_{\mathcal{T}}$ that are true in s.

For a template collection $\mathcal{J} = \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$ we may consider having global propositions $AP_{\mathcal{J}}$[3] as well as the propositions for each template type $AP_{\mathcal{T}_1}, \ldots, AP_{\mathcal{T}_n}$. With these we can then define the propositions that are true in a state $\mathbf{s}$ of $\mathcal{J}$ as

[3]Number of active processes of each type

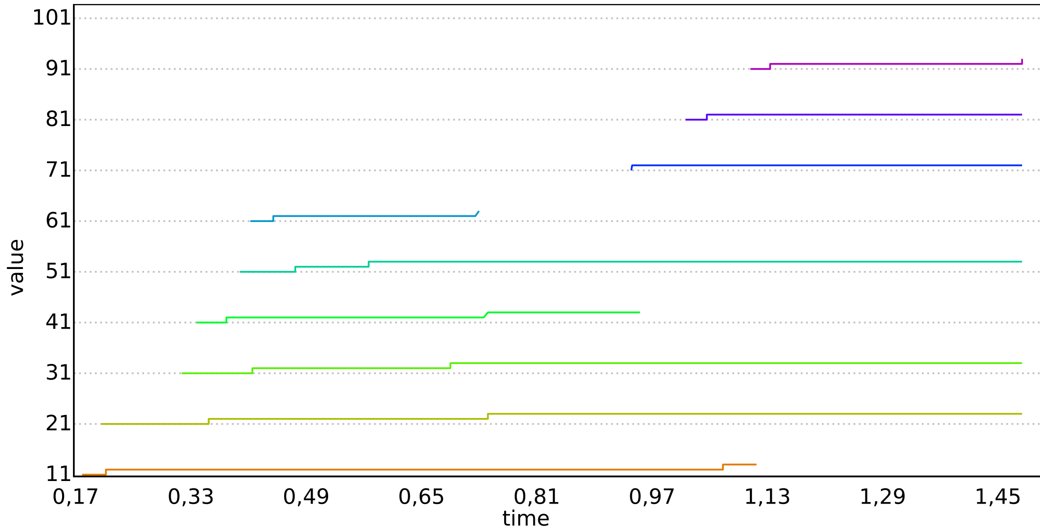

Figure 7: Gantt chart of the clients in the client-server example. Each line represent a single client. What location a client is in can be read by first calculating $\bar{y} = y \mod 10$ where after the location is given as follows: $\bar{y} = 0 \implies$ `Starting`, $\bar{y} = 2 \implies$ `Waiting`, $\bar{y} = 5 \implies$ `Working` and $\bar{y} = 8 \implies$ `Timeout`.

$$P_{\mathcal{J}}(\mathbf{s}) = \{(a, \mathcal{T}, p) | a \in \mathsf{Active} \wedge T(a) = \mathcal{T} \wedge \\ p \in P^{AP_{\mathcal{T}}}(f(a)\} \cup P'_{\mathcal{J}}(\mathbf{s}), \quad (1)$$

where $P'_{\mathcal{J}}$ gives the finite subset of $AP_{\mathcal{J}}$ that are true in s.
.

With the mapping of states to propositions we can now what we call the set of propositional runs of template collection as:

$$\Omega^{AP}(\mathcal{J}) = \{P_0 d_0 P_1 \dots | \exists \mathbf{s}_0 d_o \mathbf{s}_1 \dots \in \Omega(\mathcal{J}) \wedge \\ \forall i AP_{\mathcal{J}}(\mathbf{s}_i) = P_i\}$$

A. *Stochastic Semantics*

Following the stochastic semantics given for timed systems in [8] our stochastic semantics is based on output races among components i.e. each component chooses a delay and the one with the smallest delay wins the race. Afterwards, the winning component chooses an action to perform and another race commences.

On the component-level we associate to each state of a TIOTS $(S, s_0, \rightarrow)$ a delay density function - for a state s we write μ_s to obtain this density. In addition we assign an output probability function γ_s to all states mapping output actions to probabilities. Naturally we require that $\gamma_s(a) = 0$ if and only if $s \stackrel{a}{\not\rightarrow}$.

Now, let $\mathcal{J} = (\mathcal{T}_1, \dots, \mathcal{T}_n)$ be a template collection. We want to define a measure on a set of propositional runs of $\mathcal{J}$. The set is defined through a cylinder construction

$$\mathcal{C} = P_0 I_0 \dots P_n$$

, where $P_i \subset_{fin} AP$,

$$AP = \{(a, \mathcal{T}, p) | a \in \mathsf{PNames}, \mathcal{T} \in \{\mathcal{T}_1, \dots, \mathcal{T}_n\} \text{ and} \\ p \in AP_{\mathcal{T}}\} \cup AP_{\mathcal{J}}$$

and all I_i are non-empty intervals with rational bounds. Notice that AP is an infinite set, yet all P_i must be finite subsets.

Now, we can calculate the probability of observing a run in the cylinder $\mathcal{C} = P_0 I_0 \dots P_n$ from state $\mathbf{s} = (\mathsf{Active}, T, f)$ as

$$\mathbb{P}_{\mathcal{J}}(\mathbf{s}, \mathcal{C}) = \quad (P_{\mathcal{J}}(\mathbf{s}) \vdash P_0) \cdot \sum_{k \in \mathsf{Active}} \Big(\int_{I_0} \mu_{f(k)}(t) \cdot \\ \Big(\prod_{k' \in \mathsf{Active} \setminus \{k\}} \int_{\tau > t} \mu_{f(k')}(\tau) d\tau \Big) \cdot \\ \Big(\sum_{a \in \Sigma_o} \gamma_{f(k)^t}(a) \cdot \mathbb{P}_{\mathcal{J}}((\mathbf{s}^t)^{a/k}, C^1) \Big) dt \Big)$$

with base case $\mathbb{P}_{\mathcal{J}}(\mathbf{s}, P) = P_{\mathcal{J}}(\mathbf{s}) \vdash P$, $(P \vdash P')$ is 1 if $P = P'$ and 0 otherwise and $C^1 = P_1 I_1 \dots$. In this expression we use $\mathbf{s}^{a1!/k}$ to obtain the uniquely defined state that is reached if the process with name k performs action $a1!$.

The probability defined above requires some explanation: first it is checked if the propositions first state matches those of the cylinder, then on the outermost level we sum over all active processes. After some delay t in I_0 , the winning process chooses to perform some action. Independently, the other processes choose a delay greater than t - captured by the inner integral. Having delayed t, all the possible actions that the winning component can perform and their probabilities are taken into account. Finally, the probability of seeing the remaining part of the cylinder is multiplied.

For the remainder of this paper we let $\mathcal{C}(\mathcal{J})$ be all cylinders.

Remark 2: Allowing spawning of templates one might worry if the system will *explode* in the sense that discrete

actions may occur with shorter and shorter time between them due to growing number of components. Essentially, one might worry if the system would exhibit a zeno behaviour. Luckily it follows from Reuters criteria for birth-and-death processes[18] that if we only have exponential distributions (spanning a finite range of rates) or uniform distributions (spanning a finite range of intervals), the system will not explode. We rely on this fact as our statistical model checking algorithm requires that runs are time-diverging.

IV. Quantified Dynamic Metric Temporal Logic

In this section we present the syntax and semantics of *Quantified Dynamic Metric Temporal Logic*(QDMTL) that is highly based on MTL. The logic is defined over a template collection $\mathcal{J} = (\mathcal{T}_1, \dots, \mathcal{T}_n)$.

For any template $\mathcal{T} = (S, s_0, \rightarrow)$ we assume there exists a set of numeric expressions $Expr(\mathcal{T})$ that we can evaluate in any of its states. Similarly, we assume there exists a set of boolean expressions $Bool(\mathcal{T})$. In both cases we evaluate the expression e in state $s \in S$ by $[\![e]\!](\mathcal{T}, s)$. If $e \in Bool(\mathcal{T})$ the result is contained in the set $\{\top, \bot\}$ otherwise it returns a real-valued number. Now let $\mathbf{s} = (\mathsf{Active}, T, f)$ be a state of $\mathcal{J}$, $p \in \mathsf{Active}$ and let $T(p) = \mathcal{T}$. Then we denote the evaluation of $e \in Expr(\mathcal{T})$ in the context of p in $\mathbf{s}$ by $[\![e]\!](p, \mathbf{s}) = [\![e]\!](\mathcal{T}, f(p))$. Similar notation is used for the boolean expressions.

Assume we have a finite set of names PVar each assigned a template type. These will be placeholders for actual processes in our formulas. For $\mathtt{P} \in \mathsf{PVar}$ we denote its type $\mathcal{T}$ by $(\mathtt{P} : \mathcal{T})$. Given this set of variables, the set of numeric expressions over a template collection $\mathcal{J}$ is generated by the syntax

$$\mathcal{E} ::= c \,|\, \mathcal{E}_1 \, op \, \mathcal{E}_2 \,|\, sum(\mathcal{T})(e) \,|\, \mathtt{P}.e_2$$

where $c \in \mathbb{R}$ and $e \in Expr(\mathcal{T})$ and if $(\mathtt{P} : \mathcal{T}_1)$ then $e_2 \in Expr(\mathcal{T}_1)$. To evaluate these expressions we need to bind the process names in PVar to actual process names in PNames. We do this in terms of a mapping $M : \mathsf{PVar} \rightarrow \mathsf{PNames} \cup \{\star\}$, where $\star \notin \mathsf{PNames}$. The symbol $\star$ is here used to denote that a name has not been bound to a process. We then give the semantical meaning of an expression $\mathcal{E}$ in a given state $\mathbf{s} = (\mathsf{Active}, T, f)$ and with mapping M, denoted $[\![\cdot\mathcal{E}\cdot]\!](\mathbf{s}, M)$, recursively as:

- $[\![\cdot c \cdot]\!](\mathbf{s}, M) = c$,
- $[\![\cdot \mathcal{E}_1 \, op \, \mathcal{E}_2 \cdot]\!](\mathbf{s}, M) = [\![\cdot \mathcal{E}_1 \cdot]\!](\mathbf{s}, M) \, op \, [\![\cdot \mathcal{E}_2 \cdot]\!](\mathbf{s}, M)$,
- $[\![\cdot sum(\mathcal{T})(e) \cdot]\!](\mathbf{s}, M) = \sum_{p \in (s_\mathcal{T})} ([\![e]\!](p, \mathbf{s}))$ and
- $[\![\cdot \mathtt{P}.e \cdot]\!](\mathbf{s}, M) = [\![e]\!](M(\mathtt{P}), \mathbf{s})$

Naturally the latter is only defined if $M(\mathtt{P}) \neq \star$.

The set of QDMTL formulas for a template collection $\mathcal{J} = (\mathcal{T}_1, \dots, \mathcal{T}_n)$ is generated by the syntax

$$\varphi := \mathtt{tt} \,|\, \mathtt{P}.\tilde{b} \,|\, \mathcal{E}_1 \sim \mathcal{E}_2 \,|\, \neg\varphi \,|\, \bigcirc \varphi \,|\, \varphi_1 \wedge \varphi_2 \\ |\, \varphi_1 \, \mathrm{U}_{[a,b]} \, \varphi_2 \,|\, \mathsf{forall}(\mathtt{P} : \mathcal{T})\varphi$$

where $\mathtt{P} \in \mathsf{PVar}$, $(\mathtt{P} : \mathcal{T})$, $a, b \in \mathbb{R}_{\geq 0}$, where $a \leq b$ and $\tilde{b} \in Bool(\mathcal{T})$.

As it is custom in the family of MTL logics we use $\Diamond_{[a;b]}\varphi$ as a syntactical short hand for $\mathtt{tt}U_{[a;b]}\varphi$, $\Box_{[a;b]}\varphi$ for $\neg\Diamond_{[a;b]}\neg\varphi$. We derive the classic boolean operatorsx $\vee$ and $\implies$ in the usual way and let $\mathsf{exists}(\mathtt{P} : \mathcal{T})\varphi = \neg\mathsf{forall}(\mathtt{P} : \mathcal{T})\neg\varphi$. We call an occurrence of a subformula where every $\mathtt{P}.b$ or $\mathtt{P}.e$ is surrounded by a binding occurrence of the form $\mathsf{forall}(\mathtt{P} : \mathcal{T})$ a sentence. In the semantics, the actual binding is accomplished by updating a mapping whenever encountering an occurrence of $\mathsf{forall}(\mathtt{P} : \mathcal{T})$: if M is a mapping then

$$M[\mathtt{P} \mapsto p](x) = \begin{cases} p & \text{if } x = \mathtt{P} \\ M(x) & \text{otherwise} \end{cases}$$

Let $\omega = \mathbf{s}_0 d_0 \mathbf{s}_1 d_1 \dots$ be a run of $\mathcal{J}$, where $\mathbf{s}_i = (\mathsf{Active}_i, T_i, f_i)$ for all $i > 0$, and let φ be a QDMTL formula. Then we define satisfaction of φ with respect to a mapping M recursively as,

- $\omega \models^M tt$
- $\omega \models^M \mathtt{P}.\tilde{b}$ if $M(P) \neq \star$ and $[\![\cdot\tilde{b}\cdot]\!](M(\mathtt{P}), \mathbf{s}) = \top$
- $\omega \models^M \mathcal{E}_1 \sim \mathcal{E}_2$ if $[\![\cdot\mathcal{E}_1\cdot]\!](\mathbf{s}_0, M) \sim [\![\cdot\mathcal{E}_2\cdot]\!](\mathbf{s}_0, M)$
- $\omega \models^M \neg\varphi$ if $\omega \not\models^M \varphi$
- $\omega \models^M \bigcirc\varphi$ if $\omega^1 \models^M \varphi$
- $\omega \models^M \varphi_1 \wedge \varphi_2$ if $\omega \models^M \varphi_1$ and $\omega \models^M \varphi_2$
- $\omega \models^M \varphi_1 \, \mathrm{U}_{[a,b]} \, \varphi_2$ if there exists j such that $\omega^j \models^M \varphi_2$, $\sum_{i=0}^{j-1} d_i \in [a, b]$, and for all $k < j$, $\omega^k \models^M \varphi_1$.
- $\omega \models^M \mathsf{forall}(\mathtt{P} : \mathcal{T})\varphi$ if for each $p \in \mathsf{Active}_0$, if $T_0(p) = \mathcal{T}$ then $\omega \models^{M[\mathtt{P} \mapsto p]} \varphi$

Example 3: Consider our running example of a client-server model. Possible QMDTL formulas over this system are:

$$\Box_{0;5}(\mathsf{forall}(c : \mathtt{Client})(c.\mathtt{Waiting} \implies \Diamond_{[0;10]}(c.Working)))$$

$$\begin{aligned}&\Box_{0;5}(\mathsf{forall}(c : \mathtt{Client})(c.\mathtt{Waiting} \implies \Diamond_{[0;10]}(\\ &\qquad c.\mathtt{Working} \wedge \mathsf{exists}(\mathtt{s} : \mathtt{ServerChild}) \\ &\qquad (s.\mathtt{Working} \wedge s.id == c.id))))\end{aligned}$$

The first formula asserts that if a client within the first five time units is awaiting a connection, then it will come to the working location. The second formula asserts, in addition to this, that a `ServerChild` should also be in the `Working` location and have the same id i.e it asserts that a `ServerChild` is communicating with the client.

Definition 3: Let φ be a QDMTL sentence and let M_0 be a function where for all $\mathtt{P} \in \mathsf{PVar}$ $M_0(\mathtt{P}) = \star$. Then we define that $\omega \vDash \varphi$ iff $\omega \vDash^{M_0} \varphi$.

Theorem 1: For all QDMTL formulas φ, all template collections $\mathcal{J}$ and mappings $M : \mathsf{PVar} \rightarrow \mathsf{PNames}$, the set $\{\omega \in \Omega(\mathcal{J}) \,|\, \omega \vDash^M \varphi\}$ is measureable.

Proof: (sketch)
For this sketch we focus on the subset of QDMTL where the construction $\mathcal{E}_1 \sim \mathcal{E}_2$ is omitted. First we define the propositions per template $\mathcal{T}$ that we need to know the value of at each state. Let φ be a QDMTL formula and $\mathcal{T}$ a template then $AP_{\mathcal{T}} \subseteq Bool(\mathcal{T})$ is a finite set of properties that are relevant for φ i.e. $AP_{\mathcal{T}} = \{\tilde{b} | P.\tilde{b}$ is a sub-expression of φ and $(P : \mathcal{T})\}$. Let $\mathbf{s}$ be a state of the template collection $\mathcal{J}$ then the global propositions $AP_{\mathcal{J}}$ we are interested in are the active processes and their type thus the global propositions $AP_{\mathcal{J}} = \{(a, \mathcal{T}) | a \in \mathsf{Active} \wedge \ \mathcal{T}$ is a template$\}$. We let

$$
\begin{aligned}
P_{\mathcal{J}}(\mathbf{s}) = \{&(a, \mathcal{T}, b) | a \in \mathsf{Active} \wedge \exists P.b \in AP_{\mathcal{T}} \wedge (P : \mathcal{T}) \wedge \\
&[\![\cdot b \cdot]\!](a, \mathbf{s}) = \top\} \cup \\
&\{(a, \mathcal{T}) | a \in \mathsf{Active} \wedge T(a) = \mathcal{T}\}
\end{aligned}
$$

This is merely a concrete instance of the abstract set of proposition mapping given in Eq. (1). Notice that the propositional run induced by proposition contains enough information to conclude if a QDMTL formula is satisfied on that run [4] thus we can easily define a satisfaction relation between a propositional run and a QDMTL formula and easily show that

$$
\mathbf{s}_0 d_0 \mathbf{s}_1 \ldots \models^M \varphi \Leftrightarrow P_{\mathcal{J}}(\mathbf{s}_0) d_0 P_{\mathcal{J}}(\mathbf{s}_1) \ldots \models^M \varphi
$$

What remains is to show that the set

$$
\{\omega | \omega \in \Omega^{AP}(\mathcal{J}) \wedge \omega \models^M \varphi\}
$$

is indeed measurable i.e. is representable by a set of cylinders.

We do so by structural induction in φ. For this sketch we only show the construction for two formulas. Let

$\varphi = \mathsf{forall}(\mathtt{P} : \mathcal{T})\varphi'$

Consider the set of propositional runs:

$$
\bigcup_{P_0 I_0 \ldots P_n \in \mathcal{C}(\mathcal{J})} \{\omega = P_0 d_0 \ldots P_n \in \Omega^{AP}(\mathcal{J}) | \\
\forall i, d_i \in I_i \wedge \forall (a, \mathcal{T}) \in P_0, \omega \models^{M[\mathtt{P} \mapsto a]} \varphi'\}.
$$

Clearly this is the set satisfying φ and since it is a union of cylinders, it is measurable.

$\varphi = \mathtt{P}.\tilde{b}\mathrm{U}_{[a,b]}\mathtt{P}_2.\tilde{b}$

Consider the set of propositional runs:

$$
\bigcup_{P_0 I_0 \ldots P_n \in \mathcal{C}(\mathcal{J})} \{\omega = P_0 d_0 \ldots P_n \in \Omega^{AP}(\mathcal{J}) | \\
\exists j \text{ s.t. } \exists (M(\mathtt{P}_2), \mathcal{T}, \tilde{b}) \in P_j \text{ with } (\mathtt{P}_2 : \mathcal{T}) \\
\text{and } \forall i < j, \exists (M(\mathtt{P}), \mathcal{T}_1, \tilde{b}) \in P_i \text{ with } (\mathtt{P} : \mathcal{T}_1) \\
\text{and } (\sum_{k=1..j-1} I_k) \in [a, b]\}.
$$

Again we see this is a set composed of cylinders satisfying φ and that it is measureable since it is a union of cylinders.

■

[4] recalling that we omitted the $\mathcal{E}_1 \sim \mathcal{E}_2$

V. Statistical Model Checking of QDMTL

Statistical Model Checking [22] is a simulation based software verification technique. Underlying the technique is that the model has a *Stochastic* semantics and that we efficiently can obtain runs from its associated probability distribution. In addition we need a logic for which we can settle if a formula is satisfied by a run. Generating a run of a model and validating if a formula is satisfied gives rise to a *bernoulli* variable X that obtains the value 1 if the formula was satisfied and 0 if it was not satisfied. The probability that $X = 1$ is the probability that a random runs satisfies the formula. Let this probability be θ. Now let $X_1, X_2 \ldots X_n$ be n such bernoulli variables and let Y be a random variable obtaining the value

$$
Y = \sum_{i=1}^{n} X_i,
$$

i.e. Y counts the number of runs that satisfied the formula. Y is distributed according to a binomial distribution with succes-parameter θ. If we want to answer the *qualitative* question "is θ greater than a threshold" we may employ a *hypothesis testing* approach with a controllable level of significance[21]. In case we want to answer the *quantitative* question "what is the probability θ" we can employ an estimation approach and obtain a confidence interval. One such method is using the Chernoff-Bound as described in [8]

Statistical Model Checker

A naive statistical model checker thus consists of (1) a component that generate runs of a model, (2) a component that can settle if a formula is satisfied for a given run and (3) an algorithm from statistics that either estimates the probability θ or tests if it exceeds a threshold value.

The decoupling of the generation of the run and the validation of a run has the positive effect of making it easy to implement a statistical model checker and the individual components may easily be exchanged for others. The decoupling is, however, inefficient as time may be wasted generating a long run that violated the property after one step thus we wish to perform the validation of a run in parallel with the run generation. Previously [6] we developed such a monitoring scheme for MTL that was based on rewriting formulas: Given a run

$$
\omega = (\mathbf{s}_0, d_0)(\mathbf{s}_1, d_1) \ldots
$$

and a MTL formula φ_0 the monitor rewrites φ_0 into φ_1 using $\mathbf{s}_0$ and d_0 as input (denoted $\varphi_0 \xrightarrow{\mathbf{s}_0, d_0} \varphi_1$) in such a way that $\omega^1 \models \varphi_1$ if and only if $\omega \models \varphi_0$. Continuing to rewrite the formulas eventually transforms a formula into $\mathtt{tt}$ signalling satisfaction, or $\mathtt{ff}$ signalling violation of the property. We now provide some of the rewrite rules needed for QDMTL. The remaining rules are similar to those presented in [6].

Since we have variables in the formulas we need to take a mapping into account i.e. we rewrite tuples of the form (φ, M) where φ is a QMDTL formula and M is a mapping from PVar to PNames.

$$\frac{}{(\texttt{tt}, M) \xrightarrow{\mathbf{s},d} (\texttt{tt}, M)} \ (Atom)$$

$$\frac{[\![\cdot\mathcal{E}_1\cdot]\!](\mathbf{s}, M) = r_1 \qquad [\![\cdot\mathcal{E}_2\cdot]\!](\mathbf{s}, M) = r_2 \qquad r_1 \sim r_2}{(\mathcal{E}_1 \sim \mathcal{E}_2, M) \xrightarrow{\mathbf{s},d} (\texttt{tt}, \texttt{M})} \ (Eval_1)$$

$$\frac{[\![\cdot\mathcal{E}_1\cdot]\!](\mathbf{s}, M) = r_1 \qquad [\![\cdot\mathcal{E}_2\cdot]\!](\mathbf{s}, M) = r_2 \qquad r_1 \nsim r_2}{(\mathcal{E}_1 \sim \mathcal{E}_2, M) \xrightarrow{\mathbf{s},d} (\texttt{ff}, \texttt{M})} \ (Eval_2)$$

Above we show the most basic rewrite rules of our monitoring technique. The first rule states that if the formula is tt then this will not change due to a rewrite[5]. The two latter rules are expressing that to rewrite a comparison between expressions, the two expressions should first be evaluated and afterwards compared. If the comparison is true then the formula is rewritten into tt, otherwise it becomes ff.

Monitoring a formula $\psi = \mathsf{forall}(\texttt{P} : \mathcal{T})\varphi$ requires monitoring φ for each process of $\mathcal{T}$ i.e. we start a rewriting sequence per process - each of these rewrite sequences should have their own mapping. To denote that multiple formulas should be rewritten in parallel with each other we use the syntactical construct $\bigwedge^{+}[(\varphi_1, M_1), (\varphi_2, M_2) \ldots, (\varphi_n, M_n)]$.

The formula ψ is satisfied along a run if and only if φ was satisfied for all processes when we encountered ψ. Consequently, we rewrite $\bigwedge^{+}[(\varphi_1, M_1), (\varphi_2, M_2) \ldots, (\varphi_n, M_n)]$ into tt if all of the formulas are rewritten into tt at some point (see $ConjList_2$ below). Similarly, ψ is not satisfied if one of the processes did not satisfy φ thus if a formula is rewritten into ff - captured by the rule $ConjList_1$. Finally, if none of the above rules apply we simply rewrite the individual formulas.

$$\frac{\exists i \in \{1, \ldots, n\} \qquad (\varphi_i, M_i) \xrightarrow{\mathbf{s},d} (\texttt{ff}, M)}{\bigwedge^{+}[(\varphi_1, M_1), \ldots, (\varphi_n, M_n) \xrightarrow{\mathbf{s},d} (\texttt{ff}, M_i)]} \ (ConjList_1)$$

$$\frac{\forall i \in \{1, \ldots, n\} \qquad (\varphi_i, M_i) \xrightarrow{\mathbf{s},d} \texttt{tt}}{\bigwedge^{+}[(\varphi_1, M_n), \ldots, (\varphi_n, M_n) \xrightarrow{\mathbf{s},d} (\texttt{tt}, M_1)]} \ (ConjList_2)$$

$$\frac{\left[(\varphi_i, M_i) \xrightarrow{\mathbf{s},d} (\varphi_i', M_i')\right]_{i=1,n}}{\bigwedge^{+}[(\varphi_1, M_n), \ldots, (\varphi_n, M_n)] \xrightarrow{\mathbf{s},d} \bigwedge^{+}[(\varphi_1', M_n'), \ldots, (\varphi_n', M_n')]} \ (ConjList_3)$$

The initiating step of transforming ψ into a $\bigwedge^{+}$ construct consist of three rules(below): the first rule corresponds to immediately discovering that one of the processes did not satisfy the formula φ thus ψ is not satisfied. The second one correspond to all processes immediately satisfy φ and consequently that ψ is satisfied.

[5] An equivalent rule applies for ff

The final rule is instantiating the parallel rewrite process by first rewriting φ for all processes and then construct the conjunction between these.

$$\frac{\exists p \in \{\mathbf{s}_{\mathcal{T}}\} \qquad (\varphi, M[\texttt{P} \mapsto p]) \xrightarrow{\mathbf{s},d} \texttt{ff}}{(\mathsf{forall}(\texttt{P} : \mathcal{T})\varphi, \mathsf{M}) \xrightarrow{\mathbf{s},d} \texttt{ff}} \ (Binding_1)$$

$$\frac{\forall p \in \{\mathbf{s}_{\mathcal{T}}\} \qquad (\varphi, M[\texttt{P} \mapsto p]) \xrightarrow{\mathbf{s},d} \texttt{tt}}{(\mathsf{forall}(\texttt{P} : \mathcal{T})\varphi, \mathsf{M}) \xrightarrow{\mathbf{s},d} \texttt{tt}} \ (Binding_1)$$

$$\frac{\left[(\varphi, M[\texttt{P} \mapsto p]) \xrightarrow{\mathbf{s},d} ((\varphi_p, M_p)\right]_{p \in \mathbf{s}_{\mathcal{T}}}}{(\mathsf{forall}(\texttt{P} : \mathcal{T})\varphi, \mathsf{M}) \xrightarrow{\mathbf{s},d} \bigwedge^{+}_{\mathsf{p} \in \mathbf{s}_{\mathcal{T}}} [(\varphi_{\mathsf{p}}, \mathsf{M}_{\mathsf{p}})]} \ (Binding_3)$$

Theorem 2: Let $\omega = \mathbf{s}_0 d_0 \mathbf{s}_1 d_1 \mathbf{s}_2 d_2 \ldots$ be an infinite time-diverging run and let φ be a QMDTL sentence. In addition let M be a function where for all $\texttt{P} \in \mathsf{PVar}$ $M(\texttt{P}) = \star$. Then $\omega \vDash \varphi$ if and only if there exists a rewrite sequence

$$(\varphi, M) \xrightarrow{\mathbf{s}_0, d_0} (\varphi', M') \xrightarrow{\mathbf{s}_1, d_1} \ldots (\texttt{tt}, M'')$$

and $\omega \nvDash \varphi$ if and only if there exists a rewrite sequence

$$(\varphi, M) \xrightarrow{\mathbf{s}_0, d_0} (\varphi', M') \xrightarrow{\mathbf{s}_1, d_1} \ldots (\texttt{ff}, M'')$$

VI. EXPERIMENTS

A. Robot

We consider the imaginary example of robots moving randomly on a 2-dimensional grid in search of a specific location. In the following we describe a parameterised model so that we can make a series of experiments. Let the grid have size $x \times y$ and the goal be location $(x-1, y-1)$. Initially no robots are present on the grid but are spawned by an extra component. The robots are spawned in location $(i, 0)$ where i is randomly selected in the range $[0, x-1]$ for each robot. The robots each have a variable x and y that tells where they are on the grid and move with a rate of 2. The robots can only move up,down,left or right and only inside the grid thus they cannot move from one boundary to the other in one step.

In this random setup we may wonder how likely it is for two robots to be in the same location at once within some time limit τ. In UPPAAL SMC we can find the probability of no robots being in the same field at once by checking the query

```
Pr ([][0; τ] forall (t:Robot)
            (forall (t2:Robot)
            (t.x == t2.x && t.y ==t2.y
             imply  t==t2))).
```

We call this formula φ_1. Naturally the probability we are interested in is $1-\theta$ where θ is the probability of satisfying φ_1.

In Table I we show the probability estimated for φ_1 by UPPAAL SMC for various parameters to the model.

Setting aside that robots may actually be on the same field we might be interested in estimating the probability that the

x	y	#Robots	Probability	Time (s)
2	2	1	$[0.95, 1.00]$	0.15
4	4	2	$[0.00, 0.07]$	1.01
8	8	4	$[0.00, 0.08]$	1.06
64	64	4	$[0.41, 0.51]$	2.55
64	64	8	$[0.06, 0.16]$	2.93

Table I: Verification results for φ_1 for various grid sizes and number of robots with $\tau = 100$. In the probability column is a 95% confidence interval satisfying φ_1.

first robot reaching the goal, within τ time units, does so with a margin of δ to the second robot reaching the goal. This is straightforwardly expressed in UPPAAL SMC by the query

```
Pr ([][0;τ]forall{t:Robot} (t.x!=xgoal
        || t.y!=ygoal ∨
        ([][0;δ] forall{t2:Robot}
        ((t2.x==xgoal && t2.y==ygoal)
         imply t==t2))))
```

where $(\texttt{xgoal}, \texttt{ygoal})$ is the goal location. We call this formula for φ_2 Again we show the verification result for various parameters to the model in Table II.

x	y	#Robots	Probability	Time (s)
2	2	1	$[0.95, 1.00]$	0.17
4	4	2	$[0.00, 0.08]$	1.20
8	8	4	$[0.48, 0.58]$	3.70
64	64	4	$[0.95, 1.00]$	3.97
64	64	16	$[0.95, 1.00]$	13.81
64	64	32	$[0.95, 1.00]$	13.85

Table II: Verification results for φ_2 for various grid sizes and number of robots with $\tau = 100$ and $\delta = 1$.

The latter two results in Table II may seem surprising as one would expect the probability to decrease when adding more robots. Verifying the property

```
Pr[<=100] (<> numOf (Robot)>16)
```

which estimates the probability that more than 16 robots is spawned within the first 100 time units reveals why this is not the case. The probability of spawning more than 16 robots lie in the interval $[0.01, 0.03]$ with a confidence of 95% thus it is unlikely that 16 robots are spawned and as result even more unlikely that 32 are spawned.

B. *Client-Server*

We now turn our focus towards applying QDMTL to the client-server example. For a start, we consider the property that all clients get a connection within 10 time units after their initial connection request. We verify this within the first 10 time units of a run. In UPPAAL SMC the property is expressed as

```
Pr ([][0;10] forall { c : Client}
        (!(c.Waiting && c.retries==0) ∨
        (<>[0;10]c.Working )))
```

Ports	Probability	Time	Largest	σ
10	$[0.17, 0.27]$	17m56.750s	2054	2079.53
13	$[0.84, 0.94]$	4m55.545s	491	854.49
15	$[0.94, 1.00]$	2m11.51s	198	440.1

Table III: Probability that client do not get a connection in their first try. The `Ports` column is the number of simultaneous connections the server can handle. The Probability column is a 95% confidence interval, time is the time used for the verification in seconds. Largest is the average largest intermediate formula encountered during a verification and σ is its standard deviation

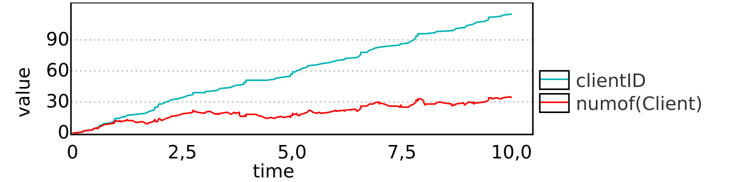

Figure 8: Development of the number of Clients. The `clientID` curve is the total number of spawned clients - the `numOf(Client)` is number of active clients.

This probability depends on the number of simultaneous connections that the server can handle - in our model this corresponds to the number of ports. In Table III we show the 95% confidence intervals obtained by verifying the property for three different number of ports. We also present the time used for the verification. In all of the three cases, the verification used 738 runs. During the generation of the runs we measured the size of the intermediate formulas, in terms of the number of terminals and recorded the largest formula encountered. In the *Largest* column we give the average of these and in the σ column we show the standard deviation. We observe that the probability is higher when we provide the server with more ports. At first, the verification time seems high. However, recall that each discrete step in the model results in an expansion of the forall and as a large large number of clients are spawned, as seen in Fig. 8, this results in large formulas. The monitoring technique is recursive in the formulas parse tree thus large formulas result in large verification time. We conjecture that the verification is smaller when the probability is high simply because the intermediate formulas encountered during verification is smaller. This is supported by the results in Table III.

If we take a closer look at the model, we notice that it is possible for the `ServerChild` to get stuck. If it chooses a long delay > 5 before signalling a connection to the client, the `Client` would have timed out and might not be ready for receiving the synchronisation from `ServerChild`. In the model there is no time out on the server side thus the `ServerChild` would wait for a disconnect synchronisation forever. This is possible in the model but verifying the property

```
Pr (<>[1;10] exists { s : ServerChild}
            ([][0;20] s.Working))
```

in UPPAAL SMC, with 10 ports available, shows that the probability of having a `ServerChild` in the `s.Working` location for 20 time units is in the interval $[0.00, 0.01]$ with a confidence of 95% thus it seems like an improbable event.

VII. Conclusion

In this paper we have presented the logic QDMTL. The logic has been developed with the specific aim of reasoning on the behaviour of the dynamic systems i.e. systems consisting of a dynamically evolving set of processes. The logic is complemented by an on-the-fly monitoring technique that given a time diverging run of the system is guaranteed to terminate and provide a correct result. We have applied our new logic to two examples, one being several robots moving autonomously on a grid and the other being our running example of a client-server architecture.

In the future we will continue our work towards providing a tool set for reasoning on dynamic systems. Especially we will extend the modelling formalism towards dynamic generation of communication channels and for allowing processes to communicate these to each other.

References

[1] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular model checking for ltl(mso). In *CAV*, volume 3114 of *LNCS*, pages 348–360. Springer, 2004.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *TCS*, 126(2):183–235, 1994.

[3] A. Bouajjani, A. Legay, and P. Wolper. Handling liveness properties in (omega-)regular model checking. In *INFINITY*, volume 138(3) of *ENTCS*. Elsevier, 2005.

[4] Abdeldjalil Boudjadar, Frits W. Vaandrager, Jean-Paul Bodeveix, and Mamoun Filali. Extending uppaal for the modeling and verification of dynamic real-time systems. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 8161 of *Lecture Notes in Computer Science*, pages 111–132. Springer, 2013. ISBN 978-3-642-40212-8.

[5] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using dy-bip. In *SC*, volume 7306 of *LNCS*, pages 1–16, 2012.

[6] Peter E. Bulychev, Alexandre David, Kim G. Larsen, Axel Legay, Guangyuan Li, and Danny Bøgsted Poulsen. Rewrite-based statistical model checking of wmtl. In *RV*, volume 7687 of *LNCS*, pages 260–275, 2012.

[7] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In *HSCC*, pages 91–100. ACM, 2010.

[8] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, Jonas van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. In Uli Fahrenberg and Stavros Tripakis, editors, *FORMATS*, volume 6919 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2011. ISBN 978-3-642-24309-7.

[9] Alexandre David, Kim Guldstrand Larsen, Axel Legay, and Danny Bøgsted Poulsen. Statistical model checking of dynamic networks of stochastic hybrid automata. 2013. to appear.

[10] José Luiz Fiadeiro and Antónia Lopes. A model for dynamic reconfiguration in service-oriented architectures. In *ECSA*, volume 6285 of *LNCS*, pages 70–85. Springer, 2010.

[11] Jasmin Fisher, Thomas A. Henzinger, Dejan Nickovic, Nir Piterman, Anmol V. Singh, and Moshe Y. Vardi. Dynamic reactive modules. In *CONCUR*, volume 6901 of *LNCS*, pages 404–418. Springer, 2011.

[12] Klaus Havelund and Kim Guldstrand Larsen. The fork calculus. *Nord. J. Comput.*, 1(3):346–363, 1994.

[13] Thomas A. Henzinger and Vlad Rusu. Reachability verification for hybrid automata. In *HSCC*, volume 1386 of *LNCS*, pages 190–204. 1998.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153271-5.

[15] Gerad J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[16] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[17] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980. ISBN 3-540-10235-3.

[18] G. E. H. Reuter. Denumerable markov processes and the associated contracting semigroups onl. *Acta Mathematica*, 97(1-4):1–46, 1957.

[19] Amir Molzam Sharifloo and Paola Spoletini. Lover: Light-weight formal verification of adaptive systems at run time. In *FACS*, volume 7684 of *LNCS*, pages 170–187, 2012.

[20] Mihaela Sighireanu and Tayssir Touili. Bounded communication reachability analysis of process rewrite systems with ordered parallelism. *ENTCS*, 239:43–56, 2009.

[21] A. Wald. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[22] Håkan L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon University, 2005.