Three-Valued Asynchronous Distributed Runtime Verification

Torben Scheffel*† and Malte Schmitz*

*Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Germany

†Graduate School for Computing in Medicine and Life Sciences, Universität zu Lübeck, Germany

***Abstract*—This paper studies runtime verification of distributed asynchronous systems and presents a monitor generation procedure for this purpose, which allows three-valued monitoring. The properties used in the monitors are specified in a logic that was newly created for this purpose and is called Distributed Temporal Logic (DTL). DTL combines the three-valued Linear Temporal Logic (LTL_3) with the past-time Distributed Temporal Logic (ptDTL), which allows to mark subformulas for remote evaluation. The monitor generation presented in this paper is based on an adopted version of the LTL_3 monitor generation, which integrates the ptDTL monitor construction. The aim of this new procedure is to increase the amount of monitorable properties compared to the properties monitorable with ptDTL. Runtime verification using this new monitoring has been implemented on LEGO Mindstorms NXT robots communicating via Bluetooth.**

I. Introduction

Software errors are an everyday problem. They occur for a variety of reasons like coding, hardware or network errors. Using software testing as a way to detect such failures requires the developer to write every check per hand. An option to check every execution path is model checking, but it is difficult to use model checking for larger systems. Model checking is especially hard to use for asynchronous distributed systems because the inherent nondeterminism leads to a significantly higher number of states than in other systems.

Another verification technique is runtime verification (RV), which checks during runtime if the behaviour of a system meets certain properties. These properties are generally specified as formulas of a temporal logic and then transformed into monitors that check the property during execution, as implemented in JavaMop in [1]. Thus RV doesn't need a complete model of the system and because of that it can be used as a verification technique for asynchronous distributed systems. One possible way to do that is by using a centralized monitor that collects the runs of all agents and then checks a property. But centralized RV for asynchronous distributed systems generates some serious overhead in communication because all the information has to reach the central monitor. Thus this paper studies RV for asynchronous distributed systems without a central but with many decentralized monitors. These monitors communicate by attaching data to the messages already sent by the agents, such that no new messages will be needed for the monitoring, which reduces the overhead even further.

RV for asynchronous distributed systems is a sparsely studied field. For synchronous distributed systems a monitoring algorithm was given in [2], which uses the Linear Temporal Logic (LTL, [3]), but this algorithm can't be used for asynchronous systems because it is based on a synchronous bus. An approach to do RV on such systems with the past-time Distributed Temporal Logic (ptDTL) was given in [4]. The logic ptDTL was invented with two-valued semantics for monitoring safety properties. Monitorability as defined in definition 15 of [5] requires that a formula is only monitorable if every finite prefix of a word can be extended such that the formula is evaluated to a final truth value. This means that it has to be possible to get to a final statement about the state of the system for every prefix of the word. Consider the properties "*Proposition a always holds.*" and "*Proposition a eventually holds.*" as an example. The first property is only monitorable using a two-valued logic if $\bot$ is the final truth value. In this case $\top$ only indicates that the property is not yet violated. The second property would not be monitorable with such a logic, even if it can be expressed, because $\top$ has to be the final truth value and $\bot$ indicates that the property is not yet fulfilled. To monitor properties of the former kind the logic has to be restricted to formulas where $\bot$ is final. The same applies to properties of the latter kind and $\top$.

It follows directly that two-valued semantics like the one from ptDTL can only be used to monitor either safety *or* guarantee properties because either $\top$ or $\bot$ can be the final truth value. However safety and guarantee properties are only a small part of all properties. Manna and Pnueli present in [6] a hierarchy of temporal properties containing the safety and guarantee classes as well as four other classes. In short, generally all properties in the safety class can be violated but not fulfilled ultimately and all guarantee properties can be fulfilled but not violated ultimately. If looked at the two examples given above, the first property is a typical safety-property and the second one a typical guarantee-property.

The next section gives examples on properties of the other classes that are interesting for RV, too, but not monitorable with ptDTL.

A. Motivating examples

As mentioned before, with ptDTL either safety *or* guarantee properties are monitorable. For example consider a property like "*It should never happen on agent a_1 that its sensor s_1 recognizes a high brightness value.*" or a property "*It always has to happen that if a sensor s_2 on agent a_2 recognizes a low brightness value, then the sensor s_1 of agent a_1 never recognized a high brightness value.*" Both are safety properties and the first property can be expressed in ptDTL as $@_{a_1} \boxminus \neg p$ where p is a proposition that indicates whether s_1 recognizes a high value and the second as $@_{a_2} \boxminus (q \rightarrow @_{a_1} \boxminus \neg p)$ where p is as before and q indicates whether s_2 recognizes a low value. Here the @ is used in ptDTL to refer to another agent in the system. For example in the last mentioned property the $@_{a_2}$ is used to denote that the whole formula should hold on agent a_2 and the $@_{a_1}$ denotes that $\boxminus \neg p$ should hold on agent a_1. In this manner properties can be spread among many agents in the system.

Thus with the @-operator ptDTL already has an appropriate tool to specify properties of different agents in one formula, but as mentioned before, ptDTL has only two different output values. It is necessary to know before the monitoring starts which output values of a logic indicate a final statement about the state of the system, for example that the system has a failure. If all possible output values would indicate such a final statement, then in every step of an agent a success or a violation of the formula would be assumed. Thus at least one value is needed that does not indicate such a final statement. In the context of two output values this leads to the need of at least one non-final value and thus if a step has this value as output, the output of the next step could be the other value. This leads to the previously mentioned statement that either safety *or* guarantee are monitorable with ptDTL because two final truth values would be needed to recognize the violation of a safety formula or the fulfilment of a guarantee formula. Furthermore using ptDTL one has to know before the monitoring starts if safety or guarantee properties are used because the appropriate output value has to be chosen as meaningful. To avoid these problems a logic with three output values is needed of which two are final to express fulfilment and violation.

Furthermore there are more important properties outside of safety and guarantee that can be verified in asynchronous distributed systems. Consider a property like "*On agent a_1 the value of the sensor s_1 has to be high until the button b on agent a_2 was pressed.*" This property is expressible but not monitorable with ptDTL because the property can be fulfilled or violated once and for all. Thus two final truth values are needed that indicate a final fulfilment or violation of the formula. Else, with the two truth values from ptDTL, either the fulfilment or the violation couldn't be recognized. With the Distributed Temporal Logic (DTL) newly developed in this paper, which is a combination of ptDTL and the three-valued Linear Temporal Logic over finite prefixes (LTL_3, [7]), the property can be expressed as $@^{\text{ft}}_{a_1} p \mathcal{U} @^{\text{pt}}_{a_2} r$ where p is as before and r indicates whether the button b was pressed on agent a_2. ft and pt stands for the formula consisting of future time (ft) or past time (pt) operators. Because DTL has a three-valued semantics with two final truth values, the property can be monitored with DTL according to [5] as shown later.

A more complex example would be a property like "*The sensor s of agent a_3 should not be activated until on agent a_2 the motor m was turned on and before on agent a_1 the sensor s' was activated.*" As before this can't be monitored with ptDTL but if it is represented with DTL as $@^{\text{ft}}_{a_3}(\neg s_1 \mathcal{U} @^{\text{pt}}_{a_2}(m \wedge \ominus @^{\text{pt}}_{a_1} \diamondsuit s'))$ the property can be monitored because of the evaluation of the until with the LTL_3 semantics. This formula is used again to show RV on asynchronous distributed LEGO Mindstorms NXT robots in section V.

As we have seen, with only two possible output values there are problems when monitoring safety and guarantee properties at the same time and that there are additional properties outside of safety and guarantee that one wants to verify as well. These examples motivate a logic that uses the ptDTL as well as the LTL_3 semantics which we will introduce as DTL.

B. Overview

This paper presents a monitor generation procedure that generates distributed monitors based on a property specified in the logic DTL, which is an especially for this purpose designed combination of ptDTL and LTL_3. The @-operator that already exists in ptDTL is also used in DTL for specifying subformulas that are properties about the run of a remote agent, so called remote properties. We redefine ptDTL and extend it to the three-valued DTL to be able to monitor boolean combinations of safety and guarantee properties where $\top$ and $\bot$ are final truth values, which requires "?" as a third truth value.

This paper is organized as follows: In section II the model of an asynchronous distributed system which is used in this paper is described. In section III a redefined version of ptDTL and a monitor generation procedure for ptDTL formulas is given. In section IV, the new monitor generation procedure is presented. Beforehand, DTL, which is used for this procedure, is defined. The implementation of RV with DTL on LEGO Mindstorms NXT robots and the performed benchmarks on this hardware are finally described in section V.

II. Asynchronous Distributed Systems

We adopt the model of distributed systems presented in [4] which is a collection of n agents $a_1, a_2, \ldots a_n \in A$ and extend it by a more formal presentation and the concept of messages. Every agent knows all other agents and is able to send messages directly to any of them. Such a message needs some finite time to arrive which may vary. The agents run concurrently and don't know anything about the internal states of the other agents except the information from the messages they receive. The computation of such a distributed system is modelled abstractly as a tuple containing a linear sequence of states for every agent. Every state contains information about the internal state of the agent, the messages sent by this

agent and the messages received by this agent since its last state. As the content of the messages doesn't matter such a message is represented only by its sequence number $i \in \mathbb{N}$. Every agent increments its sequence number each time it sends a message.

For every agent $a \in A$ there exists a set AP^a with the atomic propositions that may hold in a state of this agent and a set M^a of message symbols representing incoming and outgoing messages of this agent. The message symbols are defined as follows

$$M^a = \{\uparrow_i^b, \downarrow_i^b \mid b \in A \backslash \{a\}, i \in \mathbb{N}\}$$

where $\uparrow_i^b \in M^a$ means that agent a sends the message with sequence number i to agent b and $\downarrow_i^b \in M^a$ represents agent a receiving the message with sequence number i coming from agent b. A state of agent a is then defined as an element of the alphabet

$$\Sigma^a = 2^{\mathrm{AP}^a \cup M^a}$$

and a run of an agent a is a word $w^a = w_0^a w_1^a w_2^a \ldots \in (\Sigma^a)^\omega$ with prefixes $w_{0..k}^a = w_0^a w_1^a \ldots w_k^a \in (\Sigma^a)^*$. A run of the whole system is an n-tuple $w = (w^{a_1}, w^{a_2}, \ldots, w^{a_n}) \in \Sigma^\omega$ of runs for the n agents of A where we abuse notation $\Sigma^\omega = (\Sigma^{a_1})^\omega \times (\Sigma^{a_2})^\omega \times \ldots \times (\Sigma^{a_n})^\omega$ for the set of all possible runs. Such a run $w \in \Sigma^\omega$ of the system contains no further information about the temporal relation of the states of the different agents. Every linearization satisfying the constraint that sending a message with sequence number i from an agent a to an agent b occurs before receiving that message at the agent b is allowed. For further usage we will introduce the function last_a returning the latest position that agent a is aware of in w^b for a given remote agent b.

A. Monitoring Asynchronous Distributed Systems

For the purpose of doing RV in such an asynchronous distributed system we attach monitors to the agents. These monitors work together to check a property of the distributed system, but they only communicate by adding some data to the messages already sent by the agents. They cannot force their agent to send a message or even communicate on their own. This design is chosen to minimize the overhead created through the monitoring. Therefore messages are not sent only for the monitoring but the messages already sent by the system for other purposes are reused for the monitoring.

Every agent keeps track of the states of the remote monitors in a knowledge vector and adds information on all the current outputs of monitors that are needed by monitors attached to other agents. Figure 1 shows an example of the intended information flow between the monitors attached to different agents. With every incoming message the knowledge vector is updated.

In a system with m monitors attached to the agents the set of possible assignments of a knowledge vector is defined as $V = \{1, 2, \ldots, m\} \to \mathbb{B}$ for an arbitrary truth domain $\mathbb{B}$. Let $v^a \in V$ be an assignment of a knowledge vector of agent $a \in A$ then $v^a(i)$ contains the last output of the monitor μ_i that agent a is aware of. The knowledge

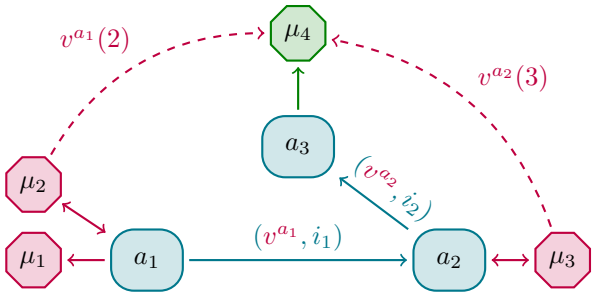

Fig. 1. Example of the information flow between cooperating monitors. The monitor μ_4 needs information regarding the output of the monitors μ_2 and μ_3 attached to remote agents. Agent a_1 first sends a message with sequence number i_1 to agent a_2 and adds the output of its monitor contained in its knowledge vector v^{a_1} to it. The knowledge vector of agent a_2 is updated with the newer information $v^{a_1}(2)$. Next a_2 sends a message with sequence number i_2 to a_3 and adds its knowledge vector v^{a_2} to the message. This way the monitor μ_4 receives information about the output of μ_2 and μ_3 in the knowledge vector as $v^{a_2}(2)$ and $v^{a_2}(3)$.

vector of an agent is updated with every incoming message if the message contains newer output values of the remote monitors than the ones currently stored in the knowledge vector. This way information can be handed on from one monitor to another even if a direct message between the corresponding agents was never sent. This solution continuously computes the best possible approximation to a global state of the system. The used concept is inspired by Lamports algorithm for generating a global snapshot (see [8]) of a distributed system. Lamport timestamps (see [9]) or vector clocks (see [10]) can be used to make sure that the value in a knowledge vector is only overwritten with newer values even if not all messages are delayed for the same amount of time.

In the following sections we will define logics for such asynchronous distributed systems consisting of n agents $a_1, a_2, \ldots, a_n \in A$. The semantics will be defined on a formula and an execution $w = (w^{a_1}, w^{a_2}, \ldots, w^{a_n}) \in \Sigma^+$ of the system where Σ^+ is the set containing all possible executions of the system. The procedure described to distribute information about the state of remote agents through the system will be taken into account in the semantics using the function $\mathrm{last}_{a_1} : \Sigma^+ \times A \times \mathbb{N} \to \mathbb{N}$ defined as follows.

Definition 1 (Last known position)**.** *For an execution $w \in \Sigma^+$ of the system, two different agents $a, b \in A$, a position g in the execution w^a of agent a and a position h in the execution w^b of agent b we have*

$$\mathrm{last}_b(w, a, h) = g$$

iff an interleaving sequence of $m \leq n$ agents and $m-1$ message sequence numbers $a_1, i_1, a_2, i_2, \ldots, a_{m-1}, i_{m-1}, a_m$ exists such that information is passed on through the messages from agent $a = a_1$ to agent $b = a_m$. To ensure that information can be passed on we require that

$$\forall j < m : \exists k : \uparrow_{i_j}^{a_{j+1}} \in w_k^{a_j} \wedge \exists \ell : \downarrow_{i_j}^{a_j} \in w_\ell^{a_{j+1}}$$

and for the first message $\uparrow_{i_1}^{a_2} \in w_g^a$ with g maximal and for the last message $\downarrow_{i_{m-1}}^{a_{m-1}} \in w_k^b$ with $k \leq h$ holds. If both agents are the same we use the convention $\mathrm{last}_a(w, a, h) = h$.

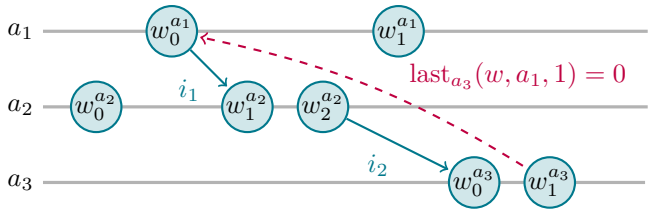

Fig. 2. Execution $w \in \Sigma^+$ of a distributed system. State $w_0^{a_1}$ in position 0 is the last state of the execution w^{a_1} of agent a_1 that agent a_3 is aware of in state $w_1^{a_3}$ in position 1 of its execution w^{a_3}.

In conclusion $\text{last}_b(w, a, h) = g$ means that agent b in position h of its execution w^b knows exactly the information about agent a that already existed in position g of its execution w^a because a sequence of messages exists which passed on this information from a to b. An example is given in figure 2.

Using the last known position defined above we are now able to define the known prefix of a run or an execution of an asynchronous distributed system. The prefix known to an agent b at a position of its run contains the prefixes of the runs or executions of every agent, each up to the last position known by agent b.

Definition 2 (Known prefix)**.** *For a run or an execution $w \in \Sigma^\infty$, an execution $u \in \Sigma^+$, an agent $b \in A$ and a position h in the execution w^b of agent b we have*

$$\text{known}_b(w, h) = u = (u^{a_1}, u^{a_2}, \ldots, u^{a_n})$$

such that $\forall a \in A : u^a = w^a_{0..k}$ if $k = \text{last}_b(w, a, h)$ is defined.

This definition leads to the known prefix being undefined just for the words of the agents that we don't know anything about.

A monitor can only use information from monitors attached to remote agents if the information is contained in the known prefix. Everything happening on a remote agent after the last known position has not yet reached the current agent. The following sections will refer back to the known prefix to define the semantics of the @-operator. The known prefix has to be taken into account in the semantics to make sure that monitors can be constructed for the distributed logics presented in this paper.

III. Past-Time Distributed Temporal Logic

The past-time Distributed Temporal Logic (ptDTL) is a logic for specifying properties of asynchronous distributed systems. It was invented by Sen et al. in [4] and is based on the past-time Linear Temporal Logic (ptLTL). Besides the operators from ptLTL, ptDTL has an @-operator which is used to spread a property over different agents.

Definition 3 (ptDTL syntax adopted from [4])**.** *Let $p \in \text{AP}^a$ be an atomic proposition of the agent $a \in A$ belonging to the innermost @ surrounding p and $a_1, a_2 \in A$ be agents. Then the set of* ptDTL *formulas is inductively defined as follows:*

$$\begin{array}{lllll}
\chi ::= & @^{\text{pt}}_{a_1}\psi & & & \\
\psi ::= & \text{true} \mid p \mid \psi \vee \psi \mid \ominus\psi & \mid & \\
& \text{false} \mid \neg p \mid \psi \wedge \psi \mid \circledcirc\psi & \mid & \\
& \psi \,\mathcal{S}\, \psi \mid \diamondsuit\psi \mid \neg\psi \mid \psi \rightarrow \psi & \mid & \\
& \psi \,\mathcal{T}\, \psi \mid \boxminus\psi \mid @^{\text{pt}}_{a_2}\psi & & &
\end{array}$$

The set of ptDTL formulas including subformulas is obtained by directly starting with the nonterminal ψ and is called sub(ptDTL).

The operator $\ominus\psi$ should be read "previously ψ", $\circledcirc\psi$ "weak previously ψ", $\psi \,\mathcal{S}\, \psi'$ "ψ since ψ'", $\diamondsuit\psi$ "eventually in the past ψ", $\psi \,\mathcal{T}\, \psi'$ "ψ triggers ψ'", $\boxminus\psi$ "always in the past ψ", and $@^{\text{pt}}_{a_2}\psi$ "ψ at agent a_2 using past semantics". The formal ptDTL semantics is defined next. We adopt the general idea from Sen et al. in [4] but use our model of the distributed system and the known prefix as given above to define the semantics of the @-operator. Thus the semantics of the @-operator is defined more precisely.

Definition 4 (ptDTL semantics adopted from [4])**.** *The ptDTL semantics for an execution $w = (w^{a_1}, w^{a_2}, \ldots, w^{a_n}) \in \Sigma^+$ of a distributed system are defined through the relation $\models_a \subset \Sigma^+ \times \mathbb{N} \times \text{sub(ptDTL)}$ for every agent $a \in A$ with $b \in A$ being another agent, $p \in \text{AP}^a$ a proposition, $i \in \mathbb{N}$ a position and $@^{\text{pt}}_a\varphi, @^{\text{pt}}_a\psi \in$ ptDTL formulas as follows:*

$$\begin{array}{ll}
w, i \models_a \text{true} & \\
w, i \models_a p & \text{iff } p \in w^a_i \\
w, i \models_a \neg\varphi & \text{iff } w, i \not\models_a \varphi \\
w, i \models_a \varphi \vee \psi & \text{iff } w, i \models_a \varphi \text{ or } w, i \models_a \psi \\
w, i \models_a \ominus\varphi & \text{iff } w, i-1 \models_a \varphi \text{ and } i > 0 \\
w, i \models_a \varphi \,\mathcal{S}\, \psi & \text{iff } \exists k \leq i : w, k \models_a \psi \\
& \text{and } \forall k < \ell \leq i : w, \ell \models_a \varphi \\
w, i \models_a @^{\text{pt}}_b\varphi & \text{iff } u \models_b \varphi \text{ and } u^b \text{ def.} \\
& \text{or } w_0 \models_b \varphi \text{ and } u^b \text{ undef.} \\
& \text{with } u = \text{known}_a(w, i)
\end{array}$$

using the notation $w_0 = (w_0^{a_1}, w_0^{a_2}, \ldots, w_0^{a_n})$. Furthermore, $w \models_a \varphi$ holds iff $w, |w^a| - 1 \models_a \varphi$ holds and $w \models @^{\text{pt}}_a\varphi$ holds iff $w \models_a \varphi$ holds. The same equivalences as in ptLTL *apply for the other operators.*

The semantics for the @-operator is defined based on the known prefix u of the current agent a. If u^b is defined, which means that some data about the state of the monitors from agent b reached agent a through a sequence of messages, the formula $@^{\text{pt}}_b\varphi$ holds iff $u \models_b \varphi$ holds, because u^b is the run of b at whose end the message was send. If no data from b reached a until position i of the run of a, then by convention $@^{\text{pt}}_b\varphi$ holds iff the first step of b fulfils φ.

A. Monitor generation

In this section let

$$\text{remote}(\varphi) = \{@^{\text{pt}}_{a_1}\xi_1, @^{\text{pt}}_{a_2}\xi_2, \ldots, @^{\text{pt}}_{a_m}\xi_m\}$$

be the set of remote subformulas (all subformulas starting with the @-operator) of a given ptDTL formula $@^{\text{pt}}_a\varphi \in$ ptDTL with the cardinality $m = |\,\text{remote}\,(\varphi)|$.

The generation of a monitor for $@_a^{\text{pt}}\varphi$ is done by creating a monitor for every remote subformula in remote(φ). Such a monitor has an internal state, takes one character $s \in \Sigma^{a_i}$ and outputs a truth value in $\mathbb{B}_2$. The monitor will be implemented on agent a_i and therefore only realizes the steps performed on agent a_i. The monitor has access to the current assignment $v^{a_i} \in V$ of the knowledge vector of agent a_i where $V = \{1, 2, \ldots, m\} \to \mathbb{B}_2$ with the two-valued truth domain $\mathbb{B}_2 = \{\top, \bot\}$. As described in section II-A the knowledge vector is updated with every incoming message if the message contains newer output values of the remote monitors than the ones currently stored in the knowledge vector. Therefore the last known monitor output of a remote subformula $@_{a_k}^{\text{pt}}\xi_k$ can be accessed as $v^{a_i}(k)$. In the implementation these values are actually handled as extra propositions but for simplicity's sake this solution is not explained in detail here.

The monitors for such formulas are algorithms that evaluate a certain ptDTL formula with every step the corresponding agent performs. The algorithm presented here is based on Sen et al. in [4] and uses a local memory containing the evaluation of all temporal subformulas in the last step. For a given formula $@_a^{\text{pt}}\varphi \in$ ptDTL the set of local temporal subformulas temporal(φ) contains all subformulas of φ with a temporal operator as outermost operator that are not inside another remote subformula. The set temporal(φ) includes φ itself if its outermost operator is a temporal one. A temporal subformula may consist of other temporal subformulas. This leads to an acyclic dependency graph of temporal subformulas of $@_a^{\text{pt}}\varphi$. For the rest of this section let $\varphi_1, \varphi_2, \ldots, \varphi_{|T|} \in T = \text{temporal}(\varphi)$ be the sequence of temporal subformulas of the formula φ treated in this section. We then have $\forall \varphi_i, \varphi_k \in \text{temporal}(\varphi) : \varphi_i \in \text{temporal}(\varphi_k) \Rightarrow i \leq k$.

Let $@_a^{\text{pt}}\psi, @_a^{\text{pt}}\psi' \in$ ptDTL be past formulas, $\Gamma^a = 2^{\text{AP}^a}$ the input alphabet of agent a and $s \in \Gamma^a$ the current input character, $q, q' \in Q = \{1, 2, \ldots, |T|\} \to \mathbb{B}_2$ two assignments of a local memory assigning a truth value to every temporal subformula and $b \in \mathbb{B}_2$ the output of the monitor. For the monitoring function $f_\varphi : Q \times \Gamma^a \times V \to Q \times \mathbb{B}_2$ the equation $f_\varphi(q, s, v) = (q', \beta)$ holds iff $\beta = \text{eval}(\varphi)$ and $q'(i)$ with $1 \leq i \leq |T|$ is inductively given as follows:

$$q'(i) = \begin{cases} \text{eval}(\psi) & \text{if } \varphi_i = \ominus\psi \\ \text{eval}(\psi' \vee (\psi \wedge q(i))) & \text{if } \varphi_i = \psi \,\mathcal{S}\, \psi' \end{cases}$$

For the formulas $@_a^{\text{pt}}\psi, @_a^{\text{pt}}\psi' \in$ ptDTL and a proposition $p \in \text{AP}^a$ we define the evaluation of a formula depending on the current memory assignment q', the memory assignment q of the last step, the current state s and the current assignment v of the knowledge vector inductively as follows:

$$\begin{aligned} \text{eval}(\text{true}) &= \top \\ \text{eval}(\neg\,\psi) &= \overline{\text{eval}(\psi)} \\ \text{eval}(p) &= \begin{cases} \top & \text{if } p \in s \\ \bot & \text{else} \end{cases} \\ \text{eval}(@_{a_i}^{\text{pt}}\xi_i) &= \begin{cases} \top & \text{if } v(i) = \top \\ \bot & \text{else} \end{cases} \end{aligned}$$

$$\text{eval}(\psi \vee \psi') = \text{eval}(\psi) \sqcup \text{eval}(\psi')$$

$$\text{eval}(\varphi_i) = \begin{cases} q(i) & \text{if } \varphi_i = \ominus\psi \\ q'(i) & \text{if } \varphi_i = \psi \,\mathcal{S}\, \psi' \end{cases}$$

The computation of the new assignment q' and the definition of eval can easily be extended to other operators using their equivalences.

In the initial assignment $q_0 \in Q$ of the local memory the strong operators $\mathcal{S}$, $\diamondsuit$ and $\ominus$ are mapped to $\bot$ and the weak operators $\mathcal{T}$, $\boxminus$ and $\odot$ are mapped to $\top$. In the implementation presented in section V we compute the first step of the monitor during the monitor generation procedure using the initial values of the propositions which are set in the source code. The outputs of the first step are used as initial values in the knowledge vector assignment v.

We denote the system of monitoring functions generated for the formula $@_a^{\text{pt}}\varphi \in$ ptDTL and all its remote subformulas $@_{a_i}^{\text{pt}}\xi_i \in \text{remote}(\varphi)$ with M_φ. Let $w \in \Sigma^+$ be an execution of the distributed system A. In this system, with every step an agent performs in the execution w, for all monitors of this agent their new state is computed using their monitoring functions. With every outgoing message the knowledge vector of the sending agent is first updated with the current outputs of its monitor and then attached to the message. With every incoming message the knowledge vector of the receiving monitor is updated with the newer data contained in the knowledge vector attached to that message. We denote the output of the main monitoring function f_φ of this system after reading the whole execution w as $M_\varphi(w)$.

The following theorem states the correctness of this system of monitoring functions.

Theorem 1 (Correctness of ptDTL monitor)**.** *Let $w \in \Sigma^+$ be an execution of the distributed system A and $@_a^{\text{pt}}\varphi \in$ ptDTL a formula that should be evaluated on agent $a \in A$. Further let M_φ be the system of monitoring functions for φ. We then have*

$$M_\varphi(w) = \top \text{ iff } w \models @_a^{\text{pt}}\varphi.$$

Proof: As the used monitor generation for past time LTL is well known (see [11]) this proof focuses on how the known prefix is computed using the knowledge vector of the agents. The ptDTL semantics refers to the known prefix for subformulas starting with an @ and the known prefix in turn is based on the last known position.

By definition the last known position $\text{last}_a(w, b, i)$ is the position of the execution of agent b where the latest message is sent that arrived on agent a until the current position i of the execution of a. The monitoring function computing the new state after reading the input at position i is provided with the knowledge vector that was updated regarding the remote agent b the last time with exactly this latest message mentioned before. Therefore the knowledge vector contains the information computed in the semantics using the known prefix.

Provided with the correct assignment of the knowledge vector the correctness of the monitoring function itself

follows from structural induction over its definition using the fixed point equivalence of all linear temporal logics. ■

IV. Distributed Temporal Logic

In this section we present the main contribution of this paper, the Distributed Temporal Logic (DTL) and the monitor generation procedure for this logic. DTL extends ptDTL as a logic for specifying properties of asynchronous distributed systems. It combines the three-valued LTL_3 semantics with the ptDTL semantics defined above. Using the three-valued semantics over the truth domain $\mathbb{B}_3 = \{\top, ?, \bot\}$ we get the advantages described in the introduction as shown later.

Definition 5 (DTL syntax)**.** *Let $p \in \text{AP}^a$ be an atomic proposition of the agent $a \in A$ belonging to the innermost @ surrounding p and $a_1, a_2 \in A$ be agents. Then the set of* DTL *formulas is inductively defined as follows:*

$$
\begin{array}{lllllllll}
\chi ::= & @^{\text{ft}}_{a_1}\varphi & | & @^{\text{pt}}_{a_1}\psi & & & & & \\
\varphi ::= & \text{true} & | & p & | & \varphi \vee \varphi & | & \bigcirc\varphi & | \\
 & \text{false} & | & \neg p & | & \varphi \wedge \varphi & | & & \\
 & \varphi \,\mathcal{U}\, \varphi & | & \Diamond\varphi & | & \neg\varphi & | & \varphi \rightarrow \varphi & | \\
 & \varphi \,\mathcal{R}\, \varphi & | & \Box\varphi & | & @^{\text{ft}}_{a_2}\varphi & | & @^{\text{pt}}_{a_2}\psi &
\end{array}
$$

In this definition $@^{\text{pt}}_{a_i}\psi$ refers to the ptDTL *syntax defined in the previous section. The set of DTL formulas including subformulas is obtained by directly starting with the nonterminal φ and is called sub(DTL). A DTL formula φ is called future or past DTL formula iff it has the form $\varphi = @^{\text{ft}}_a\varphi'$ or $\varphi = @^{\text{pt}}_a\varphi'$, respectively.*

The operator $\bigcirc\varphi$ should be read "next φ", $\varphi \,\mathcal{U}\, \varphi'$ "φ until φ'", $\Diamond\varphi$ "eventually φ", $\varphi \,\mathcal{R}\, \varphi'$ "φ releases φ'", $\Box\varphi$ "always φ", and $@^{\text{ft}}_{a_2}\varphi$ "φ at agent a_2 using future semantics". In the next definition the formal DTL semantics is given. If the formula is a past DTL formula, it is evaluated as described for ptDTL formulas in section III. If it is a future DTL formula, it is evaluated as described below.

Definition 6 (DTL semantics)**.** *Let $a \in A$ be an agent, $u \in \Sigma^+$ an execution of the distributed system with the agents $a_1, a_2, \ldots a_n \in A$ and $\Gamma^\omega = (\Gamma^{a_1})^\omega \times (\Gamma^{a_2})^\omega \times \ldots \times (\Gamma^{a_n})^\omega$ with $\Gamma^{a_i} = \text{AP}^{a_i}$ the set of all possible runs of the system without any message being sent or received. Then the DTL semantics for a property $@^{\text{ft}}_a\varphi \in$ DTL is defined through the evaluation function $[\![\cdot \models \cdot]\!] : \Sigma^+ \times \text{DTL} \rightarrow \mathbb{B}_3$ as follows:*

$$
[\![u \models @^{\text{ft}}_a\varphi]\!] = \begin{cases} \top & \text{if } \forall w \in \Gamma^\omega : [uw, 0 \models_a \varphi] = \top \\ \bot & \text{if } \forall w \in \Gamma^\omega : [uw, 0 \models_a \varphi] = \bot \\ ? & \text{else} \end{cases}
$$

where the concatenation of a finite execution $u = (u^{a_1}, u^{a_2}, \ldots, u^{a_n}) \in \Sigma^+$ and a possible extension in the form of an infinite run $w = (w^{a_1}, w^{a_2}, \ldots, w^{a_n}) \in \Gamma^\omega$ of the system is performed element-wise as $uw = (u^{a_1}w^{a_1}, u^{a_2}w^{a_2}, \ldots, u^{a_n}w^{a_n}) \in \Sigma^\omega$ and the semantic function $[\cdot, \cdot \models_a \cdot] : \Sigma^\omega \times \mathbb{N} \times \text{sub(DTL)} \rightarrow \mathbb{B}_3$ is defined for an agent $a \in A$ with $w \in \Sigma^\omega$ being a run, $b \in A$ another agent, $p \in \text{AP}^a$ a proposition, $i \in \mathbb{N}$ a position, $@^{\text{ft}}_a\varphi, @^{\text{ft}}_a\psi \in$ DTL future formulas and $@^{\text{pt}}_a\varphi' \in$ ptDTL a past formula as follows:

$$
\begin{aligned}
[w, i \models_a \text{true}] &= \top \\
[w, i \models_a p] &= \begin{cases} \top & \text{if } p \in w^a_i \\ \bot & \text{else} \end{cases} \\
[w, i \models_a \neg\varphi] &= \overline{[w, i \models_a \varphi]} \\
[w, i \models_a \varphi \vee \psi] &= [w, i \models_a \varphi] \sqcup [w, i \models_a \psi] \\
[w, i \models_a \bigcirc\varphi] &= [w, i+1 \models_a \varphi] \\
[w, i \models_a \varphi \,\mathcal{U}\, \psi] &= \begin{cases} \top & \text{if } \exists k \geq i : [w, k \models_a \psi] = \top \\ & \text{and } \forall i \leq \ell < k : [w, \ell \models_a \varphi] = \top \\ \bot & \text{if } \forall k \geq i : [w, k \models_a \psi] = \bot \\ & \text{or } \exists i \leq \ell < k : [w, \ell \models_a \varphi] = \bot \\ ? & \text{else} \end{cases} \\
[w, i \models_a @^{\text{pt}}_b\varphi'] &= \begin{cases} \top & \text{if } u \models_b \varphi \text{ and } u^b \text{ def.} \\ & \text{or } w_0 \models_b \varphi \text{ and } u^b \text{ undef.} \\ & \text{with } u = \text{known}_a(w, i) \\ \bot & \text{else} \end{cases} \\
[w, i \models_a @^{\text{ft}}_b\varphi] &= \begin{cases} \beta & \text{if } \exists \ell : u = \text{known}_a(w, \ell) \\ & \text{and } \beta = [\![u \models @^{\text{ft}}_b\varphi]\!] \in \mathbb{B}_2 \\ ? & \text{else} \end{cases}
\end{aligned}
$$

The same equivalences as in LTL *apply for the other operators.*

The function $[\cdot \models \cdot]$ corresponds to LTL in the LTL_3 semantics. Besides the additional @-operator the operators are defined as in LTL with ? as a third truth value to pass ? from a remote future DTL formula to the outer operator. The $@^{\text{pt}}$ is defined as in ptDTL in section III and the $@^{\text{ft}}$ is used for specifying remote future DTL properties. The actual position is unimportant for $@^{\text{ft}}$ because we wait for a final truth value. If there exists no message containing one, ? is returned because nothing final is known about the remote formula.

Analogously to the LTL_3 semantics another function $[\![\cdot \models \cdot]\!]$ is defined in the semantics. This function represents the evaluation of the formula for the given execution. It only returns one of the final truth values ($\top$ and $\bot$) when the formula is evaluated to that truth value for all infinite extensions of the execution by the function $[\cdot \models \cdot]$.

As mentioned in the introduction there are more interesting properties that can be verified in an asynchronous distributed system than it is possible to monitor with ptDTL. The next theorem shows that more properties are monitorable with DTL than with ptDTL.

Theorem 2 (Monitorability)**.** *The classes safety, guarantee and their boolean combinations are monitorable with DTL.*

The proof for this theorem follows directly from the analysis of the monitorability using different truth domains given in [5].

A. Monitor generation

As before the generation of monitors for a DTL formula $@^x_a\varphi$ with $x \in \{\text{ft}, \text{pt}\}$ and an agent $a \in A$ is done by

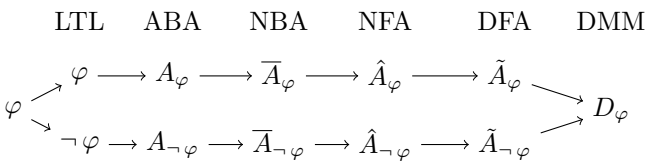

Fig. 3. LTL_3 monitor generation procedure overview (see [7]).

creating a monitor for every remote subformula. Because there are two types of remote subformulas, past and future DTL formulas, two monitor construction procedures are needed: The one for ptDTL formulas already presented in the last section and the one for future DTL formulas presented in this section.

In this section like in section III-A on the ptDTL monitor construction let

$$\text{remote}(\varphi) = \{@_{a_1}^{x_1}\xi_1, @_{a_2}^{x_2}\xi_2, \ldots, @_{a_n}^{x_n}\xi_n\}$$

with $x_i \in \{\text{pt}, \text{ft}\}$ be the set of remote subformulas of the given DTL formula $@_a^x\varphi$ with the cardinality $n = |\,\text{remote}\,(\varphi)|$. Using this indexing we define the set of possible assignments of the knowledge vector as $V = \{1, 2, \ldots, n\} \to \mathbb{B}_3$.

The generation of monitors for those formulas is based on the monitor construction for LTL_3 formulas shown in figure 3. To translate the additional @-operator the construction of the Alternating Büchi Automaton (ABA) is changed as follows. To transform a future DTL formula $@_a^{\text{ft}}\psi$ into ABAs with knowledge vector, it is first transformed into negation normal form. The ABA $A_\psi = (\Gamma^a, V, Q_\psi, \psi, \delta_\psi, F_\psi)$ is then constructed with the alphabet $\Gamma^a = 2^{\text{AP}^a}$ of agent a, the assignments V of knowledge vectors in the distributed system, the set $Q_\psi = \text{sub}(\psi)$ of states consisting of all subformulas of ψ, the initial state ψ and the set $F_\psi = \{@_{a_1}^{\text{ft}}\psi_1, \neg@_{a_1}^{\text{ft}}\psi_1, \psi_1\,\mathcal{R}\,\psi_2\} \subseteq Q_\psi$ of accepting states. For $\delta_\psi : Q_\psi \times \Gamma^a \times V \to \mathbb{B}^+(Q_\psi)$ where $\mathbb{B}^+(Q)$ is the set of positive boolean combinations of a set Q, applies the following for a proposition $p \in \text{AP}^a$, the current state $s \in \Gamma^a$, the current assignment $v \in V$ of the knowledge vector and formulas $@_{a_1}^{\text{ft}}\psi_1, @_{a_1}^{\text{ft}}\psi_2 \in \text{DTL}$:

$$\delta_\psi(p, s, v) = \begin{cases}\text{true} & \text{if } p \in s\\ \text{false} & \text{else}\end{cases}$$

$$\delta_\psi(\neg p, s, v) = \begin{cases}\text{true} & \text{if } p \notin s\\ \text{false} & \text{else}\end{cases}$$

$$\delta_\psi(\psi_1 \vee \psi_2, s, v) = \delta_\psi(\psi_1, s, v) \vee \delta_\psi(\psi_2, s, v)$$
$$\delta_\psi(\psi_1 \wedge \psi_2, s, v) = \delta_\psi(\psi_1, s, v) \wedge \delta_\psi(\psi_2, s, v)$$
$$\delta_\psi(\bigcirc\psi_1, s, v) = \text{repl}(\psi_1, v)$$
$$\delta_\psi(\psi_1\,\mathcal{U}\,\psi_2, s, v) = \delta_\psi(\psi_2 \vee (\psi_1 \wedge \bigcirc(\psi_1\,\mathcal{U}\,\psi_2)), s, v)$$
$$\delta_\psi(\psi_1\,\mathcal{R}\,\psi_2, s, v) = \delta_\psi(\psi_2 \wedge (\psi_1 \vee \bigcirc(\psi_1\,\mathcal{R}\,\psi_2)), s, v)$$

$$\delta_\psi(@_{a_i}^{\text{ft}}\xi_i, s, v) = \begin{cases}\text{true} & \text{if } v(i) = \top\\ \text{false} & \text{if } v(i) = \bot\\ @_{a_i}^{\text{ft}}\xi_i & \text{else}\end{cases}$$

$$\delta_\psi(\neg@_{a_i}^{\text{ft}}\xi_i, s, v) = \begin{cases}\text{true} & \text{if } v(i) = \bot\\ \text{false} & \text{if } v(i) = \top\\ \neg@_{a_i}^{\text{ft}}\xi_i & \text{else}\end{cases}$$

$$\delta_\psi(@_{a_i}^{\text{pt}}\xi_i, s, v) = \begin{cases}\text{true} & \text{if } v(i) = \top\\ \text{false} & \text{else}\end{cases}$$

$$\delta_\psi(\neg@_{a_i}^{\text{pt}}\xi_i, s, v) = \begin{cases}\text{true} & \text{if } v(i) = \bot\\ \text{false} & \text{else}\end{cases}$$

As before v is the current assignment of the knowledge vector that contains the truth values for every remote formula, taken from the newest message of the corresponding agent and $\text{repl} : \text{sub}(\text{DTL}) \times V \to \text{sub}(\text{DTL})$ is a function which replaces every remote future DTL formula with true or false as soon as its final truth value is available in the knowledge vector. This is required for the automaton to make use of the special behaviour of three-valued remote properties. After one final truth value is found, all later arriving messages contain the same truth value. If the future DTL formulas aren't replaced as described the satisfiability check for subformulas wouldn't notice that this truth value is the final one and therefore the remote property can be fulfilled always or never from now on. This satisfiability check, which is implicitly contained in the DTL semantics, is explicitly performed during the monitor generation by an emptiness-per-state test as described later. Thus the replacement function repl is needed in the monitor construction to be able to recognize final truth values as soon as possible in the DTL monitor.

After the ABA A_ψ for $@_a^{\text{ft}}\psi$ is constructed, it is transformed into an equivalent Nondeterministic Büchi Automaton (NBA) where V is viewed as part of the alphabet. For the alphabet Σ_ψ of the resulting NBA $\Sigma_\psi = \Gamma^a \times V$ then applies. In the same way $@_a^{\text{ft}}\neg\psi$ is transformed into an ABA $A_{\neg\psi}$ and then into an NBA as well. On these NBAs the emptiness-per-state test is performed, which tests for every state if there exists a word that is accepted when starting in this state, and only those states whose tests succeeded are marked as accepting. The resulting automata are then interpreted as Nondeterministic Finite Automata (NFA) and transformed into an equivalent Deterministic Finite Automata (DFA). Out of A_ψ the DFA $\tilde{A}_\psi$ and out of $A_{\neg\psi}$ the DFA $\tilde{A}_{\neg\psi}$ is created. From the two DFAs the Deterministic Moore Machine (DMM) D_ψ is constructed, which is the monitor. The DMM simulates both DFAs. The output for a state is $\top$ if the current state of the DFA $\tilde{A}_{\neg\psi}$ is not accepting and $\bot$ if the current state of $\tilde{A}_\psi$ is not accepting. Otherwise the output for a state is ?. The whole process from the ABAs to the DMM is described in [7].

As before in the ptDTL monitor generation the system of monitoring functions and DMMs generated for the formula $@_a^x\varphi \in \text{DTL}$ and all its remote subformulas $@_{a_i}^{x_i}\xi_i \in \text{remote}(\varphi)$ with $x, x_i \in \{\text{pt}, \text{ft}\}$ is denoted by M_φ. Again let $w \in \Sigma^+$ be an execution of the distributed system A. The new states of the monitoring functions and DMMs in this system and the assignments of the knowledge vectors of the agents are computed in the same way as described in section III-A on ptDTL monitor generation and $M_\varphi(w)$ denotes the output of the monitor generated for the main formula φ after this systems reads the whole execution w.

The following theorem states the correctness of this

system of monitoring functions and DMMs.

Theorem 3 (Correctness of DTL monitors)**.** *For an execution $w \in \Sigma^+$ of the distributed system A, a DTL formula $\varphi \in$ DTL and the corresponding monitoring system M_φ the following holds:*

1) *If $\varphi = @_a^{\text{pt}}\psi$ is a past time DTL formula, then $M_\varphi(w) = \top$ iff $w \models \varphi$.*
2) *If $\varphi = @_a^{\text{ft}}\psi$ is a future time DTL formula, then $M_\varphi(w) = [\![w \models \varphi]\!]$.*

Proof: Statement 1) follows directly from theorem 1.

To show statement 2) let $u \in \Sigma^\omega$ be a run of the distributed system A and $\hat{u}^a \in (\Gamma^a)^\omega$ the run of agent $a \in A$ contained in u without the message symbols. Based on the assumption that it will be provided with the correct knowledge vector assignment in every step the ABA A_φ generated for a formula $@_a^{\text{ft}}\varphi$ rejects $\hat{u}^a$ iff $[\![u, 0 \models_a \varphi]\!] = \bot$. For formulas using only the common LTL operators this observation results from the ABA being generated using the well known translation from linear temporal logics to an alternating Büchi automaton from [12]. For the new @-operator the observation holds because $\delta_\varphi(@_{a_i}^{\text{ft}}, s, v) =$ false is specified in the definition if $v(i) = \bot$ for any sign $s \in \Gamma^a$. As the knowledge vector is assigned using the same mechanism as in 1) the assumption of its correctness follows from 1). Performing the emptiness-per-state test described above on the NBA equivalent to this ABA we get an NFA and its equivalent DFA $\tilde{A}_\varphi$. As the output of the DMM depends only on the rejection of words in $\tilde{A}_\varphi$ and $\tilde{A}_{\neg\varphi}$ statement 2) follows from the observation above. ∎

B. Advantages

The main advantage of DTL over ptDTL are the additional future DTL formulas. Through these formulas the set of monitorable properties of DTL is much larger than the one of ptDTL. More precisely according to theorem 2 it is possible to monitor properties of the classes safety, guarantee and their boolean combinations with DTL instead of either safety *or* guarantee with ptDTL. By adding future LTL operators to ptDTL the resulting logic is more succinct than ptDTL, too. This can be compared to LTL with past being exponentially more succinct than LTL as shown in [13]. Furthermore the semantics of future DTL formulas is anticipatory and thus final truth values are recognized as soon as possible.

Another advantage comes from the usage of the model from section II as a model for asynchronous distributed systems. Through the usage of message symbols and the known prefix given there the @-operator in ptDTL and DTL is defined precisely. The semantics is defined over executions containing not only the atomic propositions but also information about the messages sent and received by the agents. Hence for the precise theoretical evaluation of a formula only the execution is needed. This isn't possible with ptDTL as defined by Sen et al. in [4] because there the communication between the agents is not modelled explicitly.

Furthermore it is possible to use future DTL formulas as remote formulas and thus truth values of those remote formulas are three-valued, too. Thus a monitor knows from some truth values in messages that they won't change anymore on the remote monitor which offers more certainty about the state of a remote formula. This is especially useful to combine the output of different monitors into one.

V. Application on LEGO Mindstorms[1]

In this section we finally present the application of distributed RV on LEGO Mindstorms NXT agents. The firmware of these agents provides access to actors like motors and lights as outputs, analog and digital sensors as inputs and allows communication between the agents via Bluetooth. The sensors are numbered from 1 to 4 and the actors from A to C. We program the agents using the C-like programming language Not eXactly C (NXC) developed by John Hansen[2]. The monitor generation is written in Scala, takes the NXC source code files as input, adds C-code to step the monitors and instruments the C-code to compute assignments of the atomic propositions.

We chose this platform to demonstrate the usage of DTL and for educational purpose. Prototypical setups can be easily built using LEGO bricks. Because we actually instrument C-like code the implementation can be adopted for many other embedded systems, e.g., industrial control systems.

To add RV the user needs to annotate the NXC code of the agents. Annotations are C-comments to allow the compilation of the code with *and* without the generated code. For every monitor a monitor annotation with a DTL formula has to be added to the code of the agent on which the monitor should run. The propositions used in the formula must be declared afterwards. Propositions can be assigned by using explicit C-functions that are evaluated each step, by using explicit assignments that are executed if the run reaches its location or by program transformation via regular expressions. Remote subformulas can be declared as external propositions that will be assigned with the output of remote monitors declared as public. Additionally annotations have to be added to step the monitor. A monitor can be explicitly stepped if the execution reaches the annotation, it can be stepped if a proposition changes or after a given time.

The preprocessing of the NXC code of the agents is done by the Scala program in several steps:

1) The software analyzes the code and parses the annotations. As the result of this step a list of all public monitors of all agents is generated. If a monitor's output is referred to an external proposition it has to be declared as public.
2) The monitors are generated using the appropriate monitor generation procedure described in sections III-A and IV-A.
3) A mapping from all public monitors to positions in the knowledge vector is computed. For every

[1]The software and a video of the case study are available at http://www.isp.uni-luebeck.de/legomeetsrv.

[2]http://bricxcc.sourceforge.net

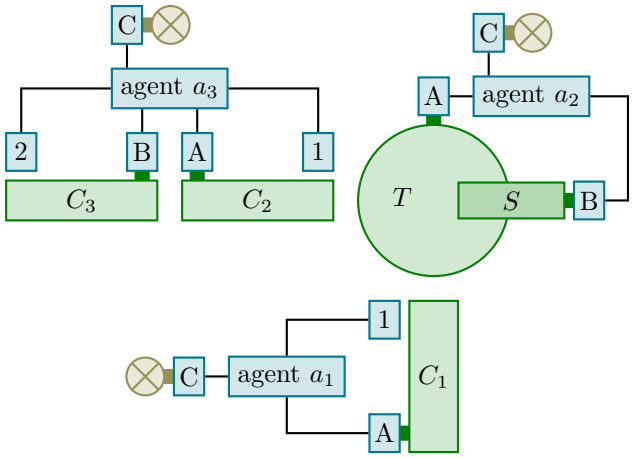

Fig. 4. Schematic layout of an assembly line consisting of three conveyors C_i, a revolving table T and a slider S. The assembly line is controlled by three NXT agents a_i. Their inputs 1, 2 and 3 are attached to brightness sensors, which detect work pieces next to them. Their outputs A, B and C are attached to motors and yellow indicator lamps.

public monitor the initial output is computed. For future DTL monitors this is ? and for past DTL monitors the output of the first step is used as initial output. To perform the first step the initial values of the declared propositions are used. This may include external propositions referring to the outputs of other public monitors.

4) The C-code for the monitors and for the propositions is generated and inserted in the agent's NXC code and the knowledge vector is declared using the initial values computed above. Two callback functions are generated. One that is called every time a message is sent and that adds the knowledge vector to the original message content and another one that is called with every incoming message and that parses the incoming message, updates the knowledge vector accordingly and returns the original message content.

A. Case Study

To demonstrate how DTL and its monitors can be used to monitor a real distributed system we choose the setup of an assembly line. Modern industrial control systems are distributed systems where the components are embedded in the part of the plant they control. Distributed RV can be used to assert the correct functionality of such a system. The following example shows how one can benefit from the DTL monitoring presented in this paper in such a use case.

Consider the setup visualized in figure 4: Three conveyors C_i, a revolving table T and a slider S move a work piece through the assembly line. Conveyor C_1 moves the work piece onto the table T, which rotates it by 90 degrees. The slider S moves the rotated work piece onto conveyor C_2, which passes it on to the output conveyor C_3. Every conveyor is controlled by the motors A or B attached to it. The brightness sensors 1 or 2 become active if a work piece is moved next to them. Output C of the agents is connected to a yellow lamp indicating the output of selected monitors.

In this setup we now want to verify that a work piece only leaves the revolving table if the table was rotated and a work piece has entered it before. This property is fulfilled or violated precisely at the moment the first work piece leaves the revolving table. Therefore a three-valued semantics is needed to monitor such a property. As already explained in the introduction the described property can be expressed in DTL as follows:

$$\varphi = @^{\mathrm{ft}}_{a_3}(\neg\, s_1 \,\mathcal{U}\, @^{\mathrm{pt}}_{a_2}(m_A \wedge \ominus @^{\mathrm{pt}}_{a_1} \diamondsuit s_1)).$$

The main monitor has to run on agent a_3, so its annotations are placed in the header of `a3.nxc`. The main monitor m_3 is defined by the LTL formula $\neg\, s_1 \,\mathcal{U}\, m_2$ and its output is displayed on the yellow light connected to this monitor via the callback `blink`. The proposition s_1 is given via a native C-function reading the value of sensor 1. The proposition m_2 is defined as external proposition whose value is assigned to the output of the monitor with the same name located on agent a_2.

```
//= MONITOR m3 FDTL = !s1 U m2 CALL blink
//= PROPOSITION s1 DEFINE Sensor(S1)
//= PROPOSITION m2 EXTERNAL a2
```

In reality further annotations are needed to set up that the monitor is stepped each time the value of a proposition changes.

In the header of the code `a2.nxc` of agent a_2 the public monitor m_2 is defined by the ptLTL formula $m_A \wedge \ominus m_1$. The proposition m_A is declared to become true when the output A was activated using a call like `OnFwd(OUT_A, 100);` and false when the output A was disabled using a call like `Off(OUT_A);`. The proposition m_1 is defined as external like m_2 above.

```
//= PUBLIC MONITOR m2 PTDTL = ma && (*) m1
//= PROPOSITION ma ON /OnFwd\(OUT_A[^)]+\);/ ↩
                 OFF /Off\(OUT_A\);/
//= PROPOSITION m1 EXTERNAL a1
```

Finally in the header of the code `a1.nxc` of agent a_1 the public monitor m_1 is defined by the ptLTL formula $\diamondsuit s_1$ and the proposition s_1 is declared as before.

```
//= PUBLIC MONITOR m1 PTDTL = <*> s1
//= PROPOSITION s1 DEFINE Sensor(S1)
```

The examples above show how distributed RV can be added to an existing distributed system using preprocessing of the source code following annotations given in comments. To do so in a real application one not only needs to specify the property to verify but also how the propositions should be assigned based on the internal state of the agent under scrutiny and when the state transitions of the monitors take place.

B. Benchmarks

Another setup is used to generate quantifiable statements on the performance of the implemented system on

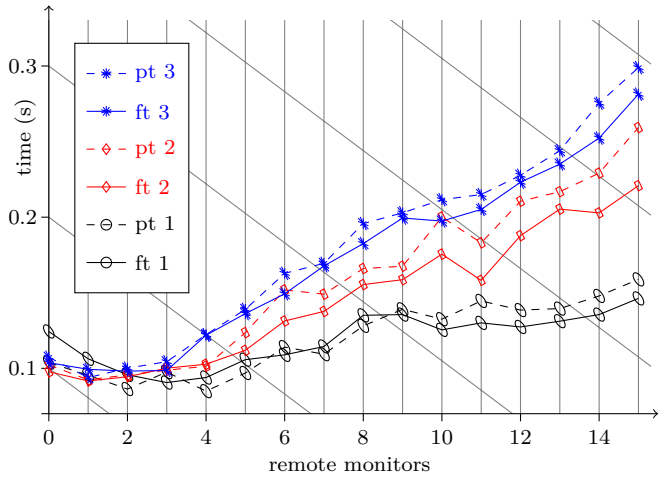

Fig. 5. Average round trip time of a message sent from a main agent to a remote agent and back. A future time (ft) or past time (pt) monitor is attached to the main agent using external propositions referring to public monitors attached to the remote agent. These monitors use 1, 2 or 3 atomic propositions switched in every iteration.

LEGO Mindstorms NXT robots. Instead of four only two agents $a, b \in A$ are used. The main agent a sends a message `ping` and waits for the answer `pong` of the remote agent b. This is repeated 100 times in one pass and the time between sending the first `ping` and receiving the last `pong` is measured to compute the average round trip time of one such message pair.

This system is monitored using formulas of the following form

$$@_a^x(\varphi_0 \,\mathcal{U}(\varphi_1 \,\mathcal{U}(\ldots \mathcal{U}(\varphi_{i-1} \,\mathcal{U}\, \varphi_i)))$$

where $x \in \{\mathrm{pt}, \mathrm{ft}\}$ is varied to compare both techniques. In case of pt of course the operator $\mathcal{S}$ is used instead of $\mathcal{U}$. The formulas φ_i are of the same principle form $@_b^{\mathrm{pt}}(p_0 \,\mathcal{S}(p_1 \,\mathcal{S}\, p_2))$ using the atomic propositions $p_i \in \mathrm{AP}^b$ whose assignments are switched in every iteration to maximize the computation needed to step the monitors. These formulas are chosen as their size can be flexibly varied and they generate most complicated monitors.

Figure 5 shows the results of these benchmarks. The round trip time of one message pair is plotted over the total number of remote monitors. On the main agent a one formula consists of a maximum of seven remote formulas which means that for every seventh remote formula a new main formula is created. The benchmark results reveal that the monitors for future time formulas are not slower than the monitors for past time formulas, which mainly use the algorithm presented in [4]. One can conclude that the DTL monitors presented in this paper are more powerful in the sense of monitorability without creating more monitoring overhead than the ptDTL monitors.

VI. Conclusion

We developed a monitor generation procedure for three-valued RV of asynchronous distributed systems. With this procedure we are able to monitor many properties that weren't monitorable before on such systems. In particular this includes boolean combinations of safety and guarantee properties. Besides that, DTL monitors detect final verdicts as soon as possible using the implicit satisfiability check performed by the emptiness-per-state test in the LTL_3 monitor construction procedure.

Furthermore we implemented the DTL monitor generation as a code preprocessing on LEGO Mindstorms NXC source code. The implementation is done fully in Scala and provides an easy to use interface. Of course this includes the ability to monitor a local property specified in ptDTL or LTL_3. We used this platform as an easy to program model of asynchronous distributed systems to demonstrate distributed RV on real world systems and to generate benchmarks. The implementation can be extended to any other embedded system programmed in a C-like language with little changes. The benchmarks show that our extension of ptDTL only adds more features and doesn't increase the computational overhead through monitoring.

We have seen from the example of the assembly line that monitoring with DTL is especially useful to monitor remote past time properties embedded in a local three-valued formula or for combining multiple remote three-valued properties in a local one.

References

[1] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An Overview of the MOP Runtime Verification Framework," *STTT*, vol. 14, no. 3, pp. 249–289, 2012.

[2] A. K. Bauer and Y. Falcone, "Decentralised LTL Monitoring," vol. 7436, pp. 85–100, 2012.

[3] A. Pnueli, "The Temporal Logic of Programs," in *18th FOCS*. IEEE Computer Society, 1977, pp. 46–57.

[4] K. Sen, A. Vardhan, G. Agha, and G. Rosu, "Efficient Decentralized Monitoring of Safety in Distributed Systems," in *26th ICSE*, A. Finkelstein, J. Estublier, and D. S. Rosenblum, Eds. IEEE Computer Society, 2004, pp. 418–427.

[5] Y. Falcone, J.-C. Fernandez, and L. Mounier, "What can you Verify and Enforce at Runtime?" *STTT*, vol. 14, no. 3, pp. 349–382, 2012.

[6] Z. Manna and A. Pnueli, "A Hierarchy of Temporal Properties," in *PDOC*, C. Dwork, Ed. ACM, 1990, pp. 377–410.

[7] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[8] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[9] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[10] C. J. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," *Aust. Comput. Sci. Commun.*, vol. 10, no. 1, pp. 56–66, 1988.

[11] K. Havelund and G. Rosu, "Synthesizing monitors for safety properties," in *8th TACAS*, ser. Lecture Notes in Computer Science, J.-P. Katoen and P. Stevens, Eds., vol. 2280. Springer, 2002, pp. 342–356.

[12] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Banff Higher Order Workshop*, F. Moller and G. M. Birtwistle, Eds. Springer, 1995, pp. 238–266.

[13] N. Markey, "Temporal Logic with Past is Exponentially More Succinct," *Bulletin of the EATCS*, vol. 79, pp. 122–128, 2003.