Repeatable Decentralized Simulations for Cyber-Physical Systems

Christophe Reymann, Mohammed Foughali, Simon Lacroix

Repeatable Decentralized Simulations for Cyber-Physical Systems

Christophe Reymann*, Mohammed Foughali*, and Simon Lacroix*
*LAAS-CNRS, Université de Toulouse, France
firstname.lastname@laas.fr

***Abstract*—Simulation is very helpful for the development of cyber-physical systems, as it enables testing functionalities and their integration without full hardware deployment. For complex systems, such as fleets of heterogeneous robots, multiple simulators dedicated to particular physical processes must be interconnected, so as to build a wholesome simulation and test the overall system. A key property to ensure is that the overall simulation is repeatable. We propose a lightweight distributed architecture for time management, allowing to easily deploy complex simulations while strictly ensuring repeatability. A formal model of the architecture is provided, along with a proof of progress. An open source implementation, with a binding to the robotic ROS framework is made available.**

I. Introduction

A. On repeatability

Repeatability (*aka* replicability) in simulation can be defined as: *the same initial conditions should always produce the same simulation results*. Repeatability is a cornerstone of scientific simulation: it is a basic (although not sufficient [1]) requirement to obtain verifiable results. If the simulation depends on the system load or the network latency, it may produce non-realistic results that go unnoticed by the user.

Furthermore, repeatability plays a key role in cyber-physical systems development, as it allows to reproduce bugs and perform regression testing. It also enables launching batches of simulations without worrying about the system load, results being identical even in resource starvation scenarios where all simulations cannot run concurrently.

B. Distributed simulations

Complex cyberphysical systems are built compositionally: they integrate various software components ranging from close to the hardware (*e.g.* controllers) to more abstract components (*e.g.* reasoning and supervision tasks). Naturally, the simulation in this context also relies on compositionality. Indeed, numerous simulators dedicated to a given domain of physics may be required, ranging from generic (*e.g.* graphics rendering for vision simulation), to more specialized (*e.g.* terramechanic simulators for wheels/soil interactions). To simulate a wholly integrated software system, we need sound techniques to connect and reuse these simulators.

The need for simulation in the integration of multiple actors has led to the development of comprehensive international *standards* such as HLA [2] and DIS [3], designed to integrate a distributed series of simulators.

Fig. 1 shows a very simple abstraction of a distributed simulation in layers.

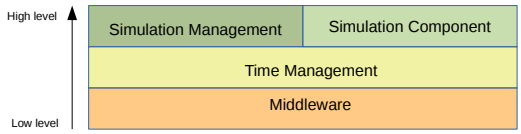

Fig. 1. Architecture of a distributed simulation

- At the core is the capability to exchange messages through a *middleware*.
- The *time management* layer is used to synchronize the simulators and enforce global consistency of the simulation.
- *Simulation management* refers to managing the whole life process of the simulation, *e.g.* setup and monitoring.
- *Simulation components* are built on top of this software stack.

Not shown here is the *common object models* (i.e. data structures with appropriate semantics), necessary for the intercommunication between components and usually described using an *interface description language*.

In the following paragraphs, we briefly introduce the simulation components and time management layers.

a) Simulation components: A simulation component follows one of the two main models in distributed simulation: *discrete event simulation (DES)* and *continuous simulation (CS)*. Typically, physical components of the system, described using differential equations, follow the CS model, while input and output variables are updated and broadcasted periodically using a fixed *time step* for integration, following a DES model. The simulator interface could then also be described by a DES model. Only a limited set of interactions between objects (*e.g.* interactions with third-party systems in the environment) are non-periodic discrete events.

b) Time management: There exists multiple time management modes, which can best be described by the relation between *simulated time* and *physical time*.

- *Real time*: the simulated time flows exactly as physical time.
- *Linear time*: the simulated time flows linearly with respect to real time, using a *speedup* coefficient.
- *Non-linear time*: the simulated time flows non-linearly, it can pause for arbitrary periods before resuming, and in some situations even go back in time (see IV-A).

Most simulators only support real and linear time, while HLA and DIS both support non linear (monotonic) time. Real-time simulation is best for validation of the software integration within the tested system (in particular using *Hardware-In-*

The-Loop simulation). Non-linear time is necessary to perform repeatable simulations, but calls for a time management layer.

C. Proposition

We focus on *time management*, a necessary step toward repeatability in distributed simulations. Time management is in charge of advancing simulation time in each of the involved simulators. To yield repeatability, it must support the non-linear-time mode, ensure that messages are processed and sent in timestamp order, and prevent message losses.

Time management in non real-time mode is usually performed in a *centralized* manner: all messages from individual simulators are exchanged through a central node, which performs the necessary computations to enforce time consistency. This is the case for both HLA and DIS (Sect. IV). Centralized time management induces an additional cost on resource usage. Also, it goes against the spirit of popular decentralized middlewares used in robotics, such as ROS [4], making it difficult to adopt in this community.

We propose a fully decentralized approach of time management, seamlessly bridged with heterogeneous cyber-physical software, in which robots may take part. The approach is formally founded and satisfies important properties in the context of repeatable distributed simulations, mainly the progress of involved simulators.

Note that sound time management is necessary for repeatability, but not sufficient to ensure it. Indeed, time management architectures are not designed to solve problems arising from non-deterministic behaviors, induced by *e.g.* complex multi-threaded interactions, which are classically considered by the developer of the components.

II. DSAAM: A DECENTRALIZED TIME MANAGEMENT ARCHITECTURE

a) Overview: Our architecture, named DSAAM for "Decentralized Synchronization Architecture for Asynchronous Middleware", can be easily implemented on top of existing cyberphysical systems components. It manages time in a completely decentralized manner, and satisfies the *Time consistency* essential property: messages are emitted and processed in a deterministic fashion, and components must wait for each others messages in order to advance. This guarantees repeatability of the simulation if other non-deterministic behaviors, not related to time management, are correctly handled.

b) Characteristics of a DSAAM System: To enforce time consistency, the following constraints are imposed to the underlying simulation components:

- *Time-stamped messages*: all exchanged messages must be time-stamped. The semantics of the timestamp is that each message represents a piece of the world state at the simulation time indicated by its timestamp.
- *Periodicity*: messages are sent with a timestamp period that is known, which tells at which simulation time to expect the next message – this period may vary.

The last constraint is a strong one. For example, it forbids request-response mechanisms that may be triggered at arbitrary points in simulation time. In Sect. V-B, we see why this constraint is necessary and discuss how it could be loosened.

A DSAAM system is made of a collection of simulators encapsulated in nodes, that exchange messages through *flows*. *Flows* have always one source, and any number of sinks, as depicted in Fig. 2. Every exchanged message includes a *timestamp* and a validity *period*. Timestamps and periods can be any kind of variable that belong to a totally ordered set endowed with addition (we use the natural integers $\mathbb{N}$).

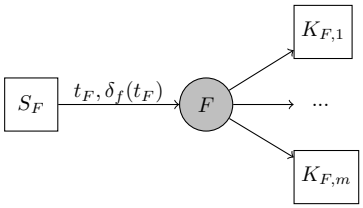

Fig. 2. A flow F with source node S and sinks $K_{F,1}...K_{F,m}$, with next emitted message timestamp t_F and period $\delta_f(t_F)$.

The period represents a contract between sources and sinks: if a sink receives a message with timestamp t and period δ, then the *next* message on this flow will have timestamp $t + \delta$. It is very similar in nature to the notion of lookahead (Sect. IV-A2), but instead of providing a lower bound it provides the *exact* timestamp of the next message. The period δ may change between each message.

c) Properties of a DSAAM node: To satisfy the constraints above, and therefore ensure time consistency, the following four rules on consumption and emission of messages inside a node must be enforced:

1) Messages are consumed by increasing timestamp order, no matter the source they are coming from.
2) Emitting a message with timestamp T forbids future consumption of messages with timestamp $T' < T$.
3) Consuming a message with timestamp T forbids future emission of messages with timestamp $T' \leq T$
4) No incoming message can be lost or discarded before it is consumed by the simulation.

III. FORMAL FOUNDATIONS AND GUARANTEES

We first present transition systems, the formalism on which operational semantics of DSAAM are based. We give syntactical definitions, and then derive operational semantics that unambiguously specifies the behavior of DSAAM systems in line with the requirements/properties/rules in Sect. II. Finally, we rely on the model to prove important properties of DSAAM systems.

A. Preliminaries

1) Transition System TS:

a) Syntax: A TS is a tuple $\langle U, Q, q_0, \longrightarrow \rangle$ where:

- U is a finite set of implicitly typed variables. We use $\mathrm{dom}(u)$ to denote the domain of variable $u \in U$,
- Q is a set of states. Each state is an interpretation of each $u \in U$ to a value $q(u) \in \mathrm{dom}(u)$,
- $q_0 \in Q$ is the initial state that maps each variable to its *initial* value,
- $\longrightarrow$ is a set of transitions. Each transition $t \in \longrightarrow$ is a binary relation that defines for every state $q \in Q$ a (possibly empty) set of successors $t(q) \subseteq Q$. We write $q \xrightarrow{t} q'$ iff $q' \in t(q)$.

b) Semantics: The evolution of a TS is subject to *taking enabled transitions*. A transition $t \in \longrightarrow$ is enabled iff TS is at state q and $t(q) \neq \emptyset$. After taking t, TS reaches a state q' in $t(q)$. We may thus define the set of *reachable* states $Q_r \subseteq Q$: a state q is reachable, *i.e.* $q \in Q_r$, iff there exists a (possibly empty) sequence of transitions σ such that $q_0 \xrightarrow{\sigma} q$.

2) Transition Diagram TD: We define a graphical notation for a TS (called a Transition Diagram TD) and a composition operation between TDs. The composition of multiple TDs (viewed as components) results in a TS (viewed as the *system*).

a) Syntax: A TD C (component) is a finite directed graph with V its set of vertices and E its set of edges. C operates on a finite set of variables, X. The vertex v_0 in V is the unique initial vertex of C. If e connects vertex v_a to vertex v_b, then we may write $v_a \xrightarrow{e(g_e, op_e)} v_b$ where (i) g_e is a boolean expression over X and (ii) op_e an atomic sequence of operations over variables in X (op_e is said *side-effect free* on variable $x \in X$ iff $op_e(x) = x$). In this paper, tautology guards and side-effect-free operations are not represented.

Fig. 3 shows a simple TD example with two vertices, v_0 (initial, denoted with a sourceless incoming edge) and v_1, and two edges e and e'. Guards (in green) and operations (in red) are over the set of variables $X = \{x_1, x_2\}$ with $dom(x_1) = dom(x_2) = \mathbb{N}$.

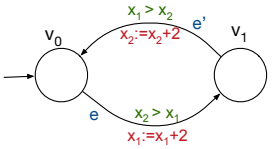

Fig. 3. A TD example

b) Semantics: Let $q(g)$ denote the truth value of guard g at state q, and $q'_{|Y} = op(q_{|Y})$ denote that the valuation of each variable $y \in Y$ at state q' agrees with the result of op over y from state q ($q'(y') = q(y')$ if op is side effect free on some $y' \in Y$). We can then associate to any TD C a TS $\langle U, Q, q_0, \longrightarrow \rangle$ that gives its semantics (Sect. III-A1), where:

- $U = X \cup \pi$ where π denotes the current vertex of C ($\mathrm{dom}(\pi) = V$ and the initial value of π is v_0),
- Q is the set of states, each state is an interpretation of π and each variable in X,
- q_0 is the mapping associating π to v_0 and each variable in X to its initial value,
- $\longrightarrow$ is the set of transitions resulting from mapping each edge e in E to a transition t_e in $\longrightarrow$ as follows. If $v_a \xrightarrow{e(g_e, op_e)} v_b$ then $q' \in t_e(q)$ iff:

$$\begin{cases} (1)\ (q(\pi) = v_a \wedge q'(\pi) = v_b) \wedge \\ (2)\ q(g_e) \wedge \\ (3)\ (q'_{|X} = op_e(q_{|X})) \end{cases}$$

c) Properties: A TS satisfies the *progress* property iff there is an enabled transition at each reachable state, that is: $\forall q \in Q_r\ \exists t \in \longrightarrow : t(q) \neq \emptyset$. Progress is crucial in multi-node simulation (Sect. III-C). For example, progress is satisfied by the TS associated to the TD in Fig. 3 iff $q_0(x_2) - q_0(x_1) = 1$.

3) Composition of Transition Diagrams:

a) Through shared variables: Syntax:

The (asynchronous) parallel composition of a finite number of TDs, $C_1, \ldots, C_n$, over a set of shared variables, U_s, is denoted $\{Init\}\left[\underset{i \in 1..n}{||} C_i\right]$ where $Init$ is the function that defines for each u in U_s its initial value in $dom(u)$.

By means of compositionality, edges of different components are always distinct: if e is an edge in C_i then it cannot be an edge in C_m with $i \neq m$. Each TD C_i operates a set of local variables, denoted U_i, besides the variables in U_s ($U_i \cap U_s = \emptyset$ and $U_i \cap U_m = \emptyset$ for all indexes $i, m \in 1..n$ with $i \neq m$). Besides, each component C_i has a variable π_i to store its current vertex (Sect. III-A2). Therefore, the set of variables declared in the TS is $U = U_s \cup \left(\bigcup_{i \in 1..n} U_i\right) \cup \left(\bigcup_{i \in 1..n} \{\pi_i\}\right)$.

Semantics: Given the parallel composition $\{Init\}\left[\underset{i \in 1..n}{||} C_i\right]$, we can define a TS $\langle U, Q, q_0, \longrightarrow \rangle$ that will give the semantics of the system where U is the set of variables defined above and:

- Q is the set of states, each state is an interpretation of each variable in U,
- q_0 is the mapping associating (i) π_i to v_0^i (the initial vertex of C_i) and each u in U_i to its initial value, for each C_i and (ii) each u in U_s to its initial value $Init(u)$,
- $\longrightarrow$ is the set of transitions resulting from mapping each edge e in E_i for each component C_i to a transition t_e in $\longrightarrow$ as follows. If $v_a^i \xrightarrow{e(g_e, op_e)} v_b^i$, then $q' \in t_e(q)$ iff:

$$\begin{cases} (1)\ (q(\pi_i) = v_a^i \wedge q'(\pi_i) = v_b^i) \wedge \\ (2)\ q(g_e) \wedge \\ (3)\ (q'_{|U_i \cup U_s} = op_e(q_{|U_i \cup U_s}) \wedge \\ \quad \forall u \in U \backslash (U_i \cup U_s \cup \{\pi_i\}) : q(u) = q'(u)) \end{cases}$$

b) Adding synchronizations: Let us consider the composition above: $\{Init\}\left[\underset{i \in 1..n}{||} C_i\right]$. Let $\mathcal{E} = \bigcup_{i \in 1..n} E_i$ be the set of all edges. We define a set of *send* edges E^S and a set of *receive* edges E^R with $E^S \cup E^R \subseteq \mathcal{E}$ and $E^S \cap E^R = \emptyset$. We denote E_i^X ($X \in \{S, R\}$) the subset of edges in E^X that belong to component C_i, that is $E^X \cap E_i$. We define then the *matching* function $M : E_S \mapsto \mathcal{P}(E_R)$ ($\mathcal{P}(s)$ denotes the powerset of set s) such that, for all $i \in 1..n$, the following property is always satisfied: $\forall e \in E_i^S\ \forall e' \in M(e) : e' \notin E_i^R$. Using these definitions and notations, we extend the composition of TDs with a *strong pairwise send/receive* synchronization paradigm (see semantics below).

Semantics: The meaning of $\{Init\}\left[\underset{i \in 1..n}{||} C_i\right]$, with $E^S \cup E^R \neq \emptyset$, is the TS $\langle U, Q, q_0, \longrightarrow \rangle$ where U, Q and q_0 are the same as in Sect. III-A3a, and $\longrightarrow$ is defined as follows:

$\longrightarrow$ is the set of transitions $\longrightarrow_e \cup \longrightarrow_s$ such that:
(i) $\longrightarrow_e$ results from mapping each e in $\mathcal{E} \backslash (E_S \cup E_R)$ to the transition t_e in $\longrightarrow_e$, according to the same rules given for $\longrightarrow$ in Sect. III-A3a.
(ii) $\longrightarrow_s$ maps each pair of edges $\{e, e'\}$ s.t. $e \in E_S$ and $e' \in M(e)$ to the transition $t_{e,e'}$ in $\longrightarrow_s$ as follows. If

$v_a^i \xrightarrow{e(g_e, op_e)} v_b^i$ and $v_k^j \xrightarrow{e'(g_{e'}, op_{e'})} v_l^j$, then $q' \in t_{e,e'}(q)$ iff:

$$\begin{cases} (1)\,(q(\pi_i) = v_a^i \wedge q'(\pi_i) = v_b^i \wedge q(\pi_j) = v_k^j \wedge q'(\pi_j) = v_l^j) \wedge \\ (2)\ (q(g_e) \wedge q(g_{e'})) \wedge \\ (3)\ (q'_{|\,U_i \cup U_j \cup U_s} = op_{e,e'}(q_{|\,U_i \cup U_j \cup U_s}) \wedge \\ \quad \forall u \in (\bigcup_{\substack{m \in 1..n \\ m \notin \{i,j\}}} U_m) \cup (\bigcup_{\substack{m \in 1..n \\ m \notin \{i,j\}}} \{\pi_m\}) : q(u) = q'(u)) \end{cases}$$

where $op_{e,e'}$ denotes performing op_e **then** $op_{e'}$.

B. Formalizing DSAAM

1) Syntax:

a) Inputs: An input I is a triple $I = \langle B, T, IF \rangle$ where B is a *buffer*, T a *timestamp variable*, and IF an input interface. B is a (non-zero size) queue of *messages*, where each message m_i has a *content* c_i, a *timestamp* $t_i \in \mathbb{N}$ and a period $\delta_i \in \mathbb{N}_{>0}$ (i denotes precedence, that is m_{i+1} is the message following m_i in time, and thus $t_{i+1} = t_i + \delta_i$). Since c_i is implementation dependent (does not intervene in the semantics), we propose a simplified version of B, where each message m_i contains simply the timestamp of m_{i+1}, *i.e.* t_{i+1}. We use the following functions: *empty(B)* (resp. *full(B)*) returns true iff B is empty (resp. full), *enqueue(B,m)* (with $\neg full(B)$) returns B with message m inserted in a FIFO fashion, *first(B)* (resp. *dequeue(B)*) with $\neg empty(B)$, returns the first element of B (resp. returns B deprived from its first element).
T stores the timestamp of the next message to consume on input I, that is simply the last dequeued element from B (the first element to arrive in B when the system starts).

b) Outputs: An output O is a double $O = \langle S, OF \rangle$ where S is the timestamp of the next message to emit on O and OF the interface of O.

c) Nodes: A node N is a triple $N = \langle \mathcal{I}, \mathcal{O}, UP \rangle$ where:

- $\mathcal{I} = \{I_1, ..., I_k\}$ is a set of inputs (Sect. III-B1a),
- $\mathcal{O} = \{O_1, ..., O_l\}$ is a set of outputs (Sect. III-B1b),
- UP is a set of blackbox update operations.

Besides implementation-dependent operations (*e.g.* processing message content), UP is in charge of output timestamp updates which take part in the semantics (Sect. III-B2).

d) DSAAM System: In the remainder of this paper, we use the superscript (^{i}) to denote that an input/output belongs to node N_i. Furthermore, the elements of an input (resp. output) are uniquely defined through propagating the subscripts and superscripts of the input (resp. output) they belong to. For instance, $\langle S_j^i, OF_j^i \rangle$ are the timestamp and interface of O_j^i (the j^{th} output of node N_i). We omit the subscript/superscript when it is unimportant (*e.g.* subscripts at the level of one node). Similarly, the subscript is omitted if only the identity of the node is important (*e.g.* O^i for any output of node N_i).
A DSAAM system $\mathcal{S}$ of x nodes is thus a double $\mathcal{S} = \langle \mathcal{N}, \mathcal{F} \rangle$ where $\mathcal{N} = \{N_1, ..., N_x\}$ is a set of nodes (Sect. III-B1c) and $\mathcal{F} : \mathcal{OF} \mapsto \mathcal{P}(\mathcal{IF})$ is the flow function such that:

- $\mathcal{OF} = \bigcup_{i \in 1..x} \left(\bigcup_{j \in 1..|\mathcal{O}^i|} OF_j^i \right)$,
- $\mathcal{IF} = \bigcup_{i \in 1..x} \left(\bigcup_{j \in 1..|\mathcal{I}^i|} IF_j^i \right)$.

Therefore, we give the syntax of a DSAAM system as a reconfigurable network of reusable nodes (re-implementable in different systems by simply redefining the flow function).

e) Syntactical restrictions: In this paper, we consider only *well-formed* systems. A DSAAM system (Sect. III-B1d) is said well formed if and only if:
(1) All inputs are connected, each to one and only one output:
(i) $\mathcal{IF} \subseteq \bigcup_{i \in 1..x} \left(\bigcup_{j \in 1..|\mathcal{O}^i|} \mathcal{F}(OF_j^i) \right)$
(ii) $\forall \{OF, OF'\} \in \mathcal{P}(\mathcal{OF}) : \mathcal{F}(OF) \cap \mathcal{F}(OF') = \emptyset$
(2) A node does not send messages to itself:
$\forall OF^i \in \mathcal{OF} : IF^j \in \mathcal{F}(OF^i) \Rightarrow i \neq j$
(3) A DSAAM system forms a strongly connected component SCC.

2) Operational Semantics:

a) Nodes: The operational semantics of a node N is given over a TD (Sect. III-A2).
Vertices: $V = \{Wa, Co, Em\}$ with Wa (initial) for *waiting* and Co (resp. Em) for *consuming* (resp. *emitting*) a message.
Variables: The TD of N accesses the variables given by the syntax of N (Sect. III-B1c), that is B_i and T_i (resp. S_i) for each input I_i in $\mathcal{I}$ (resp. each output O_i in $\mathcal{O}$), see Sect. III-B1a (resp. Sect. III-B1b). Additionally, a local variable m is introduced (see below).
Edges: $E = \{be, ee, bc, ec\} \cup snd \cup recv$ such that:

- $Wa \xrightarrow{be(g_{be}, op_{be})} Em$ (begin emitting),
- $Em \xrightarrow{ee(g_{ee}, op_{ee})} Wa$ (end emitting),
- $Wa \xrightarrow{bc(g_{bc}, op_{bc})} Co$ (begin consuming),
- $Co \xrightarrow{ec(g_{ec}, op_{ec})} Wa$ (end consuming).
- $snd = \left(\bigcup_{i \in 1..|\mathcal{O}|} snd_i \right)$ such that $Em \xrightarrow{snd_i(g_{snd_i}, op_{snd_i})} Em$ (emit on output O_i) for each i in $1..|\mathcal{O}|$. This permits emitting messages at Em,
- $recv = \left(\bigcup_{i \in 1..|\mathcal{I}|} recv_i \right)$ such that $v \xrightarrow{recv_i(g_{recv_i}, op_{recv_i})} v$ for each v in V (receive on input I_i) for each i in $1..|\mathcal{I}|$. This allows receiving messages at any vertex.

To define guards and operations, we use the functions $rand(s)$ (returns randomly one element of s) and $min(s)$ (returns the smallest element of s), with s being a non empty set. We also introduce the function up, performed by the blackbox UP (syntax, Sect. III-B1c), which updates the timestamp of the next message to emit to a strictly larger value.

- **Edge be** g_{be}: $\exists i \in 1..|\mathcal{O}| \mid S_i \leq min\left(\bigcup_{j \in 1..|\mathcal{I}|} T_j \right)$,
op_{be}: $m := rand(\{i \in 1..|\mathcal{O}| \mid S_i = min\left(\bigcup_{j \in 1..|\mathcal{O}|} S_j \right)\})$.
- **Edge ee** op_{ee}: $S_m := up(S_m)$.
- **Edge bc** g_{bc}: $\exists i \in 1..|\mathcal{I}| \mid T_i < min\left(\bigcup_{j \in 1..|\mathcal{O}|} S_j \right)$,
op_{bc}: $m := rand(\{i \in 1..|\mathcal{I}| \mid T_i = min\left(\bigcup_{j \in 1..|\mathcal{I}|} T_j \right)\})$.

- **Edge ec** g_{ec}: $\neg empty(B_m)$,
op_{ec}: $T_m := first(B_m)$; $B_m := dequeue(B_m)$, which updates T_m and B_m (as explained in Sect. III-B1a).
- **Edges snd** g_{snd_i}: $m = i$.
- **Edges recv** g_{recv_i}: $\neg full(B_i)$.
This TD enforces the rules given in Sect. II at a node level. For instance, g_{be} ensures no emission begins unless the smallest timestamp is of an output (rule 2), and the variable m stores the subscript of an input/output that may consume/emit a message, that is having the smallest timestamp (rule 1).
Example: Fig. 4 shows the TD of a node with one input and two outputs (resulting from applying the rules in Sect. III-B2a). Guards and operations are simplified when possible (*e.g.* since there is only one input, op_{bc} reduces to $m := 1$). Notice how no message sending/receiving actually happens since we are, so far, only at a "component" level (no flow defined). In the following, we develop compositionally the operational semantics of a DSAAM system, where multiple nodes exchange messages, by constraining the composition of nodes TDs with synchronizations and shared variables, derived exclusively from the syntactical definition of the flow.

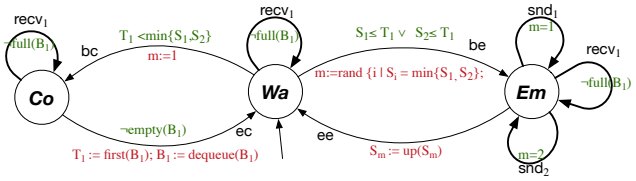

Fig. 4. A node TD example (one input and two outputs)

b) System: A DSAAM system is the parallel composition $\{Init\}\left[\underset{i\in 1..x}{||} N_i\right]$ of x node components N_i over shared variables (Sect. III-A3a), constrained with synchronizations (Sect. III-A3b). We define the synchronizations, then the set of shared variables and how the guards and operations are augmented in nodes accordingly.
Synchronizations: We derive the synchronizations from the syntax of the system (Sect. III-B1d) as follows. The set of send edges E^S (Sect. III-A3b) is the set of all snd edges in all nodes, that is $E^S = \bigcup_{i\in 1..x} E_i^S$ with $E_i^S = \bigcup_{j\in 1..|\mathcal{O}^i|} snd_j^i$. Similarly, $E^R = \bigcup_{i\in 1..x} E_i^R$ with $E_i^R = \bigcup_{j\in 1..|\mathcal{I}^i|} recv_j^i$. Now, the matching function M (Sect. III-A3b) is simply derived from the flow function $\mathcal{F}$: an edge $recv_k^l$ belongs to the set of the matching edges of an edge snd_j^i iff $IF_k^l \in \mathcal{F}(OF_j^i)$.
Shared variables: $U_s = \{Msg\} \cup \{\alpha^1, ..., \alpha^i, ..., \alpha^x\}$ is the set of shared variables (Sect. III-A3a) where Msg is the message passing variable and each α^i is used to store and update the inputs on which N_i is currently emitting. Accordingly, we enrich some edges in the nodes (Sect. III-B2a).
- On each edge be^i (edge be in each N_i), the operation $\alpha^i := \mathcal{F}(OF_m^i)$ is added to op_{be^i},
- On each edge snd^i, the guard is conjuncted with the expression $\alpha^i \neq \emptyset$ and the operation $Msg := up(S_m)$ is added,
- On each edge $recv_l^k$, the guard is conjuncted with the expression $IF_l^k \in \alpha^i$ where i is the node identity of the only output that serves I_l^k (the only output O_j^i such that $IF_l^k \in \mathcal{F}(OF_j^i)$, Sect. III-B1d and Sect. III-B1e). Each $recv_l^k$ edge is augmented with the operations $B_l^k := enqueue(B_l^k, Msg)$,
- Edge ee^i is guarded with the expression $\alpha^i = \emptyset$.
Timestamps: To start the system in a time-consistent state, we require the following in the TS of the DSAAM system: if q_0 is the initial state , then $q_0(T_j^i) = q_0(S_l^k)$ iff $IF_j^i \in \mathcal{F}(OF_l^k)$.

Only synchronizations, and guards and operations over shared variables, imply knowledge of the system, which preserves compositionality when mapping syntactical entities to their operational meanings. This "glue" between nodes enforces the rules given in Sect. II at the system level. For instance, guards on α and synchronizations between snd and $recv$ edges ensure no messages are lost or discarded (rule 4).
Example: Let us illustrate with an example. We consider the DSAAM system $\mathcal{S} = \langle \mathcal{N}, \mathcal{F} \rangle$ such that
- $\mathcal{N} = \{N_1, N_2, N_3\}$ with:
- Outputs: $|\mathcal{O}^1| = 2$ and $|\mathcal{O}^2| = |\mathcal{O}^3| = 1$,
- Inputs: $|\mathcal{I}^3| = 3$ and $|\mathcal{I}^2| = |\mathcal{I}^1| = 1$.
- The flow function $\mathcal{F}$: $\mathcal{F}(OF_1^1) = \{IF_1^2, IF_1^3\}$, $\mathcal{F}(OF_2^1) = \{IF_3^3\}$, $\mathcal{F}(OF_1^2) = \{IF_2^3\}$, $\mathcal{F}(OF_1^3) = \{IF_1^1\}$.

Using the rules given in Sect. III-B2a and Sect. III-B2b, we derive the operational semantics of $\mathcal{S}$ as the composition of TDs shown in Fig. 5. Some guards and operations are simplified on an example-dependent basis (*e.g.* the operation of be in N_2), and superscripts are removed for local variables, edges and locations (*e.g.* buffer B_1 in $op1$ is B_1^1).
Let us now explain how this works through an emission scenario, by taking the transition t_{be^1} (see mapping edges to transitions in Sect. III-A3b), and assuming the chosen output is O_1^1 ($m = 1$ after taking t_{be^1}). In this case, the message should be sent to both inputs I_1^2 and I_1^3 ($\alpha^1 = \{IF_1^2, IF_1^3\}$). Now, the enabled transitions in the system involving N_1 depend on the status of buffers B_1^2 and B_1^3. If none of them is full, one of the transition $t_{snd_1^1,recv_1^2}$ or $t_{snd_1^1,recv_1^3}$ is taken. In the former (resp. latter) case, Msg is emitted on input I_1^2 (resp. I_1^3) and IF_1^2 (resp. IF_1^3) is removed from α^1. Subsequently, $t_{snd_1^1,recv_1^2}$ (resp. $t_{snd_1^1,recv_1^3}$) is no longer enabled and the only possible transition involving N_1 is $t_{snd_1^1,recv_1^3}$ (resp. $t_{snd_1^1,recv_1^2}$) because all other transitions involving N_1 are disabled (transitions involving snd_2^1 are disabled because $m \neq 2$ and t_{ee^1} is disabled because $\alpha^1 \neq \emptyset$). Consequently, the remaining input to serve is delivered Msg by taking $t_{snd_1^1,recv_1^3}$ (resp. $t_{snd_1^1,recv_1^2}$) and α^1 becomes empty, which enables ending the emission by taking t_{ee^1}.

C. Proof of progress

DSAAM is a complex distributed system where progress (Sect. III-A2c) is a crucial property. We prove progress for all nodes in the system, which is a stricter property than progress of the system (*i.e.* absence of *deadlocks*). We start by proving the latter, then show how the former is subsequently derived.

1) Progress (system): As seen in Sect. III-A2c, a TS satisfies the progress property iff:

$$\forall q \in Q_r \ \exists t \in \longrightarrow : t(q) \neq \emptyset \qquad (1)$$

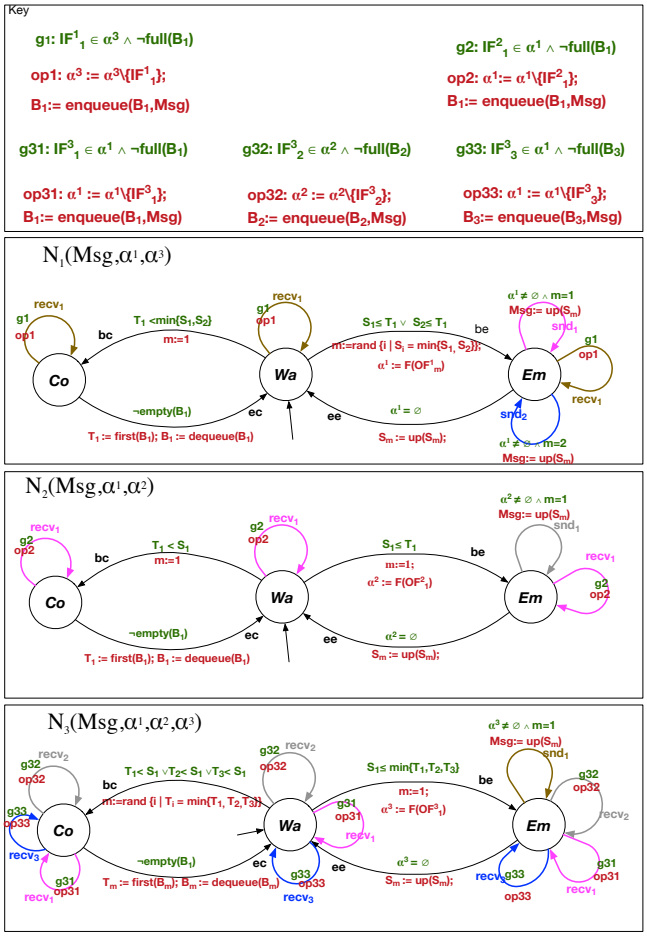

Fig. 5. Operational semantics of the DSAAM system $\mathcal{S}$. Matching send and receive edges are represented using matching colors.

Theorem 1. ***Progress (DSAAM system).***
Let Sys be a DSAAM system and TS_{Sys} the TS describing its operational semantics, with $Q_r \subseteq Q$ the set of its reachable states. TS_{Sys} satisfies the progress property (eq. 1).

Proof. We will prove **Theorem 1** by contradiction: we assume that it is false, that is a *deadlock state* q_d exists:

$$\exists q_d \in Q_r : \forall t \in \longrightarrow : t(q_d) = \emptyset \tag{2}$$

Let S^i (resp. S) be the set of output timestamp variables in node N_i (resp. in Sys), that is $S^i = \bigcup_{j \in 1..|\mathcal{O}^i|} S^i_j$ (resp. $S = \bigcup_{i \in 1..x} S^i$). Similarly, $T^i = \bigcup_{j \in 1..|\mathcal{I}^i|} T^i_j$ and $T = \bigcup_{i \in 1..x} T^i$ (for input timestamps).

Let N_k be a node with an output timestamp S^k_p that has the smallest value in S at state q_d of Q_r (in TS_{Sys}), that is at q_d:

$$S^k_p = min(S) \tag{3}$$

Now, we know that, at q_d, the current vertex π_k of the N_k TD is either Wa^k, Co^k or Em^k ($dom(\pi_k) = \{Wa^k, Co^k, Em^k\}$, Sect. III-B2a). It follows that $q_d(\pi_k) \in \{Wa^k, Co^k, Em^k\}$.

We will prove the following:

$$(a)\ q_d(\pi_k) \neq Wa^k \quad (b)\ q_d(\pi_k) \neq Co^k \quad (c)\ q_d(\pi_k) \neq Em^k$$

which results in a contradiction and thus the falseness of hypothesis 2, *i.e.* the truth of **Theorem 1**.

(a) $\boldsymbol{\pi_k \neq Wa^k}$*:* Let us assume $q_d(\pi_k) = Wa^k$. We see that $q(g_{be^k}) \vee q(g_{bc^k})$ is a tautology ($\mathbb{N}$ is a well-ordered set) and thus either t_{be^k} or t_{bc^k} is enabled. It follows that q_d is not a deadlock state, which contradicts hypothesis 2. Therefore, $q_d(\pi_k) \neq Wa^k$.

(b) $\boldsymbol{q_d(\pi_k) \neq Co^k}$*:* Let us assume $q_d(\pi_k) = Co^k$. Since q_d is a deadlock state, then $q_d(g_{ec^k})$ is false, that is:

$$empty(B^k_m) \tag{4}$$

Let O^l_i be the output connected to I^k_m, that is $IF^k_m \in \mathcal{F}(OF^l_i)$. Since B^k_m is empty, the valuation of T^k_m, holding the timestamp of the next message to consume on I^k_m, is lower-bounded by S^l_i (it is possible to dequeue $up(S^l_i)$ (op_{ec^k}) before updating S^l_i (op_{ee^l}) and $up(S^l_i) > S^l_i$).

$$T^k_m \geq S^l_i \tag{5}$$

From inequality 5 and inequality 3 we deduce:

$$S^k_p \leq T^k_m \tag{6}$$

Now, $q_d(\pi_k) = Co^k$ which means that since the last activation of t_{bc^k}, the value of T^k_m has not changed (because the only transition that modifies T^k_m is t_{ec^k}, disabled at q_d). Therefore g_{bc^k} remains true at q_d, that is:

$$T^k_m < min(S^k) \tag{7}$$

The conjuction of 7, 6 and 3 is a fallacy, and thus 4 is false which means that q_d is not a deadlock state. This contradicts hypothesis 2 and therefore $q_d(\pi_k) \neq Co^k$.

(c) $\boldsymbol{q_d(\pi_k) \neq Em^k}$*:* Let us assume $q_d(\pi_k) = Em^k$. We deduce that $S^k_m = min(S^k)$ and thus from 3:

$$S^k_m = S^k_p \tag{8}$$

From hypothesis 2, we deduce that $t_{ec^k}(q_d) = t_{snd^k_1}(q_d) = ... = t_{snd^k_{|\mathcal{O}^k|}}(q_d) = \emptyset$. After developing on these transitions, we conclude that q_d is a deadlock state at this vertex iff there is at least one receiver that N_k cannot serve because of a full buffer. That is:

$$\alpha^k \neq \emptyset \wedge (\exists i, j \in 1..x \mid IF^i_j \in \mathcal{F}(OF^k_m) \wedge full(B^i_j)) \tag{9}$$

Therefore, there is at least one message in B^i_j, which means that S^k_m is still not consumed. Also, the elements of B^i_j are inserted in a strictly monotonic order and T^i_j is the last dequeued element of B^i_j. We conclude thus that T^i_j contains a value that is strictly smaller than that of S^k_m:

$$T^i_j < S^k_m \tag{10}$$

Let us now explore the possible values of the current vertex of N_i, that is $q_d(\pi_i)$. In the following, we use m' to denote the index of the selected input in N_i (when $q_d(\pi_i) = Co^i$), to distinguish it from m, the index of the selected output in N_k.

(a') $q_d(\pi_i) \neq Wa^i$ (see (a))

(b') We assume $q_d(\pi_i) = Em^i$. There exists then $s \in S^i$ satisfying $s \leq T^i_j$. This is a contradiction because $T^i_j < S^k_p$ (10

and 8) and $S_p^k = min(S)$ (3). Therefore $q_d(\pi_i) \neq Em^i$

(c') We assume $q_d(\pi_i) = Co^i$. From hypothesis 2 we deduce:

$$empty(B^i_{m'}) \tag{11}$$

From 9 and 11 we deduce $m' \neq j$, that is node N_i is blocked consuming a message on an input different than the one N_k is trying to serve. Now, let O^z_y be the only output connected to the input $I^i_{m'}$ with buffer $B^i_{m'}$:

$$IF^i_{m'} \in \mathcal{F}(OF^z_y) \tag{12}$$

Because the buffer $B^i_{m'}$ is empty we have, following the same logic as to obtain 5:

$$T^i_{m'} \geq S^z_y \tag{13}$$

And following the same logic as to obtain 7 we have:

$$T^i_{m'} \leq T^i_j \tag{14}$$

Now, Combining 14, 13, 10:

$$S^z_y \leq T^i_{m'} \leq T^i_j < S^k_m \tag{15}$$

Conjuncting 3, 8 and 15 is a fallacy, thus $q_d(\pi_i) \neq Co^i$. Conjuncting (a'), (b') and (c') is a fallacy with $dom(\pi_i) = \{Co^i, Wa^i, Em^i\}$. It follows that 9 is false which means that q_d is not a deadlock state. This contradicts hypothesis 2 and therefore $q_d(\pi_k) \neq Em^k$.

Finally, conjuncting (a), (b) and (c) is a fallacy with $dom(\pi_k) = \{Co^k, Wa^k, Em^k\}$. It follows that hypothesis 2 is false and therefore theorem 1 holds. □

2) Progress (node): We have proven the progress of the DSAAM System as a whole, therefore there are no deadlock states in T_{Sys}. We give now a high-level succinct pseudo-proof (for the sake of readability) on how the progress of T_{Sys} implies the progress of each TD N_i involved in it.

Let N_d be a *dead* node, that is a node that violates the progress property. The system Sys is well formed, which means that each input of each node N is connected to a node different from N (Sect. III-B1e). It follows that there is at least an input of N_d connected to a different node N_i. Now, since N_d is dead, and knowing that timestamp variables in S and T increase in a strictly monotonic way, N_i will be eventually deadlocked at vertex Em^i trying to serve N_d, deadlocked. Which implies that N_i is also dead. Inductively, we reiterate the same reasoning for N_i, then for N_j such that N_j sends messages to N_i etc. Combined with the fact that a DSAAM system forms an SCC, we arrive at the conclusion that all TDs in T_{Sys} are deadlocked, which contradicts **Theorem 1** that we already proved valid in Sect. III-C1.

IV. Related work

The two widely used international standards HLA and DIS that allow to set up large-scale distributed simulations are monolithic in nature: they define everything from communication to time and simulation management, as well as domain specific models for objects (planes, ships, etc...) and algorithms (e.g. dead reckoning). A key difference lies in how messages are dispatched in the system. Other standards like the *Functional Mockup Interface* FMI [5] rise the interest of the formal community [6] for their cosimulation capacities.

Compared to these standards, our approach ensures full decentralization. Indeed, in HLA all simulators messages are exchanged through the central Run Time Infrastructure process (chapter 8 of [2]). DIS is decentralized only in the real-time mode. When switching to non real-time simulations, it needs a central process for time management. FMI comes with no dedicated time-management capabilities, and has been only linked to centralized ones [7].

A. Time management in PDES

Parallel discrete event simulations (PDES) address the problem of the simulation execution on high performance computing platforms. Time management is much researched in this domain (*e.g.* [8], [9]), and two main approaches are considered: optimistic and conservative.

1) Optimistic methods: Pioneered by the "Time Warp" algorithm [10], optimistic methods assume that all messages arrive and are processed in order. If an event is processed out of order, the entire simulation rolls back to a previous state. Therefore, optimistic methods are substantially costly in terms of resources. Also, it is difficult to handle rollbacks by ordinary OS, which led to the emergence of operating systems dedicated to these methods, such as *Time Warp OS* [11].

2) Conservative methods: Time management enforces processing all messages between simulators in the order of their timestamps, in a deterministic fashion. Conservative methods can lead to deadlocks if the topology contains circuits.

To address this problem, some techniques make use of *null messages*, messages containing only a synchronization data but no payload, to avoid deadlocks. The Chandy-Misra-Bryant (CMB) algorithm [12], [13] is based on the declaration of a *lookahead* value L for each node, which acts as a *promise* regarding the timestamp of the next message. If the node's last message was at time T, then it promises through the sending of null messages not to send any messages to any other node before $T + L$. This method can suffer from (local) deadlocks in some topologies, and from small lookahead values leading to a large amount of null messages exchanged. To tackle these drawbacks, the authors of [14] use *"conditional events"*: each node sends with each message the probable timestamp of its next message, which is valid only provided no more incoming events with smaller timestamps will arrive in the future on the node's inputs. This helps reducing the number of null messages, but still requires the nodes to periodically broadcast some synchronization messages to all other nodes.

Our approach is close to the conditional event, but additional constraints allow to simplify the algorithms and lower the communication between nodes by getting rid of the broadcasting step. Note though that the concerns in PDES are quite different from those in distributed simulations of cyber-physical systems. Indeed, a single simulation in PDES is usually distributed on a computer or a cluster to reduce execution time, whereas our primary objective is the interconnection of heterogeneous simulators, compositionality, and repeatability.

B. In robotics

Robotic simulators (*e.g.* Gazebo [15]) have not been designed to run in a distributed fashion. As a notable exception, Morse [16] supports HLA [17]. However, in a typical usage, multiple simulators communicate with each other through Morse, while a middleware links the functional components to the simulation. Furthermore, it is difficult to conciliate HLA with most of robotic middlewares (*e.g.* ROS and YARP [18]) since the former is centralized and the latter are decentralized.

Instead of relying on external simulation frameworks, our solution consists in adding a thin simulation layer on top of the robotic one. This offers a flexible time management for robotic simulations, decentralized and lightweight, which makes it easily bindable to robotic middleware. Furthermore, it would draw the developers' attention to potential problems coming from the time flow of the simulation, by allowing all components to be aware of their taking part of a simulation. All these advantages ease building repeatable simulations for cyberphysical systems made of possibly heterogenous robots.

V. Conclusion

A. Contributions

We propose DSAAM, a decentralized approach for time management to perform distributed simulations. Based on a set of rules and constraints, it is easy to implement on top of any middleware. The proof of concept of DSAAM, implemented and tested on a UAV simulation scenario (freely accessible at https://redmine.laas.fr/projects/dsaam), shows a very limited computational overhead as opposed to centralized approaches. Relying on the fact that most simulators are step-based, it only requires each simulator to know precisely, when emitting a message, the timestamp of the next message it will emit. This constraint allows to minimize the number of synchronization messages, while guaranteeing the progress of each simulator, but prevents the integration of event-based simulators.

We advocate the need of repeatable simulations, even for complex, distributed simulation infrastructures. Repeatability is beneficial to validate algorithms, perform regression testing, to speed up the development process, and to ensure the validity of simulation results no matter the computing platform used.

A *formal model* is proposed, which allows to prove the progress of involved simulators. It also defines a clear specification of the behavior of the system, and can thus be used to check the conformity of a specific implementation, even across programming languages.

The proposed implementation is easy to use and generic with respect to middleware, keeping it very lightweight. It is built upon the ROS middleware, but it can very easily be adapted to any other middleware. It respects a clear separation of concerns, which allows to integrate directly simulation layers in an existing ecosystem and lowers the developper efforts when switching from simulation to deployment.

B. Future work

One limitation of our approach is the strong periodicity constraint. It could be loosened to enhance the expressiveness of the framework in the following ways:

a) Addition of an "observation" flow type: an output of this type must wait for messages *up to and including timestamp* T being consumed before emitting a message with timestamp T. DSAAM flows describe variables that are computed from the past state of the world, which maps well with the simulation of *actuators*. However, most *sensors* can be modeled as instantaneous: *e.g.* a camera takes a snapshot of the *current* state of the world. Using "observation" flows, *sensors* could be modeled in a more natural way in DSAAM.

b) Event-based support: the main disadvantage of the approach over other time-management solutions is the lack of support for event based simulators, for which there is no guarantee of the time at which the next event will be generated. Support for event-based simulation could however be added through *e.g.* a new *event* flow type, which will have $\delta = 0$ and a request-response mechanism akin to the one in HLA.

Relaxing this constraint comes at a cost: direct circuits of *event* or *observation* flow types may introduce deadlocks. Future work includes an extended version of the DSAAM formal model with multiple flow types, and with automatic detection of such circuits.

References

[1] C. Drummond, "Replicability is not Reproducibility: Nor is it Good Science," Tech. Rep., 2009.

[2] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)Federate Interface Specification," Tech. Rep., 2010.

[3] "IEEE standard for distributed interactive simulation-application protocols," *IEEE Standard*, vol. 1278, pp. 1–52, 1998.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[5] T. Blochwitz, M. Otter, M. Arnold, C. Bausch *et al.*, "The functional mockup interface for tool independent exchange of simulation models," in *International Modelica Conference*, no. 063, 2011, pp. 105–114.

[6] F. Zeyda, J. Ouy, S. Foster, and A. Cavalcanti, "Formalising cosimulation models," in *SEFM*. Springer, 2017, pp. 453–468.

[7] M. Awais, P. Palensky, A. Elsheikh, E. Widl, and S. Matthias, "The high level architecture RTI as a master to the functional mock-up interface components," in *ICNC*. IEEE, 2013, pp. 315–320.

[8] R. Fujimoto, "Parallel and distributed simulation," in *Winter Simulation Conference (WSC)*, 2015, pp. 45–59.

[9] S. Jafer, Q. Liu, and G. Wainer, "Synchronization methods in parallel and distributed discrete-event simulation," *Simulation Modelling Practice and Theory*, vol. 30, pp. 54–73, 2013.

[10] D. Jefferson and H. Sowizral, "Fast concurrent simulation using the time warp mechanism. part i. local control." Rand. Corp. Santa Monica, Tech. Rep., 1982.

[11] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time warp operating system," in *ACM SIGOPS Operating Systems Review*, vol. 21. ACM, 1987, pp. 77–93.

[12] R. Bryant, "Simulation of packet communication architecture computer systems." MIT, Tech. Rep., 1977.

[13] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on software engineering*, no. 5, pp. 440–452, 1979.

[14] K. Chandy and R. Sherman, "The conditional-event approach to distributed simulation," Uni. of Souther California, Tech. Rep., 1989.

[15] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *IROS*, 2004, pp. 2149–2154.

[16] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine: Morse," in *ICRA*, 2011, pp. 46–51.

[17] A. Degroote, P. Koch, and S. Lacroix, "Integrating Realistic Simulation Engines within the MORSE Framework," in *R4 SIM, at Robotics: Science and Systems*, 2015.

[18] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: yet another robot platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006.