Repeatable Decentralized Simulations for Cyber-Physical Systems
Christophe Reymann, Mohammed Foughali, Simon Lacroix

To cite this version:
Christophe Reymann, Mohammed Foughali, Simon Lacroix. Repeatable Decentralized Simulations for Cyber-Physical Systems. International Conference on Software Quality, Reliability and Security (QRS), Jun 2019, Sofia, Bulgaria. hal-02156842

HAL Id: hal-02156842
https://hal.laas.fr/hal-02156842
Submitted on 14 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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.

- 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 $\delta_1(t_F)$.](image)

The period represents a contract between sources and sinks: if a sink receives a message with timestamp $t$ and period $\delta$, 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-A), but instead of providing a lower bound it provides the exact timestamp of the next message. The period $\delta$ 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 $(U, Q, q_0, \rightarrow)$ where:

- $U$ is a finite set of implicitly typed variables. We use $\text{dom}(u)$ 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 $q(u) \in \text{dom}(u)$,
- $q_0 \in Q$ is the initial state that maps each variable to its initial value,
- $\rightarrow$ is a set of transitions. Each transition $t \in \rightarrow$ 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 \( t \rightarrow \) 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 \( \sigma \) such that \( q_0 \tau^\sigma = q \).

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 \( v_a \xrightarrow{e, \text{op}_e} v_b \) where (i) \( g_e \) is a boolean expression over \( X \) and (ii) \( \text{op}_e \) is a side-effect free operation over \( X \). \( g_e \) is the mapping associating \( \text{op}_e \) to its initial value (viewed as the initial vertex of \( X \)), \( \text{op}_e \) is side effect free.

b) Semantics: Let \( q(g) \) denote the truth value of guard \( g \) at state \( q \), and \( q'_\gamma = \text{op}_e(q, \gamma) \) denote that the valuation of \( \gamma \), if \( \text{op}_e \) is side effect free on some \( \gamma' \in Y \). We can then associate to any TD \( C \) a TS \( \langle U, Q, q_0, \rightarrow \rangle \) that gives its semantics (Sect. III-A1), where:

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

\[
\begin{align*}
(1) \ (q(\pi) = v_a \land q'_\gamma(\pi) = v_b) \land \\
(2) \ q(g_e) \land \\
(3) \ q'_\gamma(x = \text{op}_e(q, x))
\end{align*}
\]

c) Properties: A TS satisfies the progress property iff there is an enabled transition at each reachable state, that is:
\[
\forall q \in Q_r \exists t \xrightarrow{\rightarrow} t(q) \neq \emptyset.
\]

Progress is crucial in multi-node simulation (Sect. III-C). For example, progress is satisfied for 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 \( \{\text{Init}\} \prod_{i=1}^n C_i \) where \( \text{Init} \) is the function that defines for each \( u \in U_s \) its initial value in \( \text{dom}(u) \).

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 \( U_s \). \( U_i \cap U_m = \emptyset \) for all indexes \( i, m \in 1..n \) with \( i \neq m \).

Besides, each component \( C_i \) has a variable \( \pi_i \) to store its current vertex (Sect. III-A2). Therefore, the set of variables declared in the TS is \( U = U_s \cup (\bigcup_{i=1}^n U_i) \cup \{\bigcup_{i=1}^n \{\pi_i\}\} \).

Semantics: Given the parallel composition \( \{\text{Init}\} \prod_{i=1}^n C_i \), we can define a TS \( \langle U, Q, q_0, \rightarrow \rangle \) 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 \( \pi_i \) to \( v_0^i \) (the initial vertex of \( C_i \)) and each \( u \in U_i \) to its initial value, for each \( C_i \),
- \( \rightarrow \) 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 \( \rightarrow \) as follows. If \( v_a \xrightarrow{e, \text{op}_e} v_b \), then \( q'_\gamma \xrightarrow{t_e(g)} q_b \) iff:

\[
\begin{align*}
(1) \ (q(\pi) = v_a \land q'_\gamma(\pi) = v_b) \land \\
(2) \ q(g_e) \land \\
(3) \ q'_\gamma(x = \text{op}_e(q, x))
\end{align*}
\]

b) Adding synchronizations: Let us consider the composition above: \( \{\text{Init}\} \prod_{i=1}^n C_i \). Let \( E = \bigcup_{i=1}^n E_i \) be the set of all edges. We define a set of send edges \( E^S \) and a set of receive edges \( E^R \) with \( E^S \cup E^R \subseteq E \) and \( E^S \cap E^R = \emptyset \). We denote \( E^X = \{X \in (S, R)\} \) 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 \( M : E_S \rightarrow P(E_R) \) (\( P(s) \) 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 \( \{\text{Init}\} \prod_{i=1}^n C_i \), with \( E^S \cup E^R \neq \emptyset \), is the TS \( \langle U, Q, q_0, \rightarrow \rangle \) where \( U, Q \) and \( q_0 \) are the same as in Sect. III-A3a and \( \rightarrow \) is defined as follows:

\( \rightarrow \) is the set of transitions \( \rightarrow \cup \rightarrow_s \) such that:

(i) \( \rightarrow \) results from mapping each edge \( e \in E \setminus (E_S \cup E_R) \) to the transition \( t_e \in \rightarrow \), according to the same rules given for \( \rightarrow \) in Sect. III-A3a

(ii) \( \rightarrow_s \) 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 \( \rightarrow_s \) as follows. If
\[ v^i_a \xrightarrow{\epsilon(g_e, op_e)} v^j_b \land v^j_b \xrightarrow{\epsilon(g_e, op_e)} v^j_k, \text{ then } q' \in t_{e,e'}(q) \text{ iff:} \]

\[
\begin{align*}
(1) & \ (q(\pi_i) = v^i_a \land q'(\pi_i) = v^j_b \land q(\pi_j) = v^j_k \land q'(\pi_j) = v^j_k) \\
(2) & \ (q(g_e) \land q(g_e')) \\
(3) & \ \forall u \in (\bigcup_{m \in i \subseteq n} U_m) \cup (\bigcup_{m \in i \subseteq n} \{x\}) : q(u) = q'(u) \\
\end{align*}
\]

where \( op_{e,e'} \) denotes performing \( op_e \) then \( op_e' \).

**B. Formalizing DSAAM**

1) **Syntax:**

   a) **Inputs:** An input \( I \) is a triple \( I = (B, T, IF) \) 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 \( m \) 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 \( O = (S, OF) \) 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 \( N = (\mathcal{I}, \mathcal{O}, \mathcal{UP}) \) where:

   \( \mathcal{I} = \{I_1, \ldots, I_k\} \) is a set of inputs (Sect. [III-B1a]),

   \( \mathcal{O} = \{O_1, \ldots, O_l\} \) is a set of outputs (Sect. [III-B1b]),

   \( \mathcal{UP} \) is a set of blackbox update operations.

   Besides implementation-dependent operations (e.g. processing message content), \( \mathcal{UP} \) 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 (\(^{\circ} \)) 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, \( (S_{j}^{\circ}, OF_{j}^{\circ}) \) are the timestamp and interface of \( O_{j} \) (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 \( S \) of \( x \) nodes is thus a double \( S = (N, F) \) where \( N = \{N_1, \ldots, N_x\} \) is a set of nodes (Sect. [III-B1c]) and \( F : OF \rightarrow P(\mathcal{IF}) \) is the flow function such that:

   \[ \mathcal{OF} = \bigcup_{i \in 1..x} \bigcup_{j \in 1..|O_i|} OF_{j}^{i} \]

   \[ \mathcal{IF} = \bigcup_{i \in 1..x} \bigcup_{j \in 1..|I_i|} IF_{j}^{i} \]

   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:

   \[ (i) \quad IF \subseteq \bigcup_{i \in 1..x} \left( \bigcup_{j \in 1..|O_i|} F(OF_{j}^{i}) \right) \]

   (2) A node does not send messages to itself:

   \[ (ii) \quad \forall \{OF_{i}, OF_{j}^{i}\} \in P(OF) : F(OF_{i}) \cap F(OF_{j}^{i}) = \emptyset \]

   (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 \( I \) (resp. each output \( O_i \) in \( O \)), see Sect. [III-B1a] (resp. Sect. [III-B1b]). Additionally, a local variable \( m \) is introduced (see below).

   **Edges:** \( E = \{be, ee, bc, ee\} \cup sn \cup rec \) such that:

   \[ Wa \xrightarrow{bc(g_w, op_w)} Em \quad \text{(begin emitting)} \]

   \[ Em \xrightarrow{ee(g_e, op_e)} Wa \quad \text{(end emitting)} \]

   \[ Wa \xrightarrow{bc(g_w, op_w)} Co \quad \text{(begin consuming)} \]

   \[ Co \xrightarrow{ee(g_e, op_e)} Wa \quad \text{(end consuming)} \]

   **snd:**

   \[ \text{snd} = \left( \bigcup_{i \in 1..|O_i|} \text{snd}_{i} \right) \]

   \[ Em \xrightarrow{snd(g_m, op_m)} Em \quad \text{(emit on output } O_i) \text{ for each } i \in 1..|O|. \text{ This permits emitting messages at } Em. \]

   **recv:**

   \[ \text{recv} = \left( \bigcup_{i \in 1..|I|} \text{recv}_{i} \right) \text{ such that } v \xrightarrow{\text{recv}_{i}(g_{\text{recv}_{i}}, op_{\text{recv}_{i}})} v \text{ for each } v \in V \text{ (receive on input } I_i) \text{ for each } i \in 1..|I|. \text{ This allows receiving messages at any vertex.} \]

   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 \( \mathcal{UP} \) (syntax, Sect. [III-B1c]), which updates the timestamp of the next message to emit to a strictly larger value.

   - **Edge be** \( g_{bc}: \exists i \in 1..|O| \quad |S_i| \leq \min \left( \bigcup_{j \in 1..|I|} T_j \right) \).

   \[ op_{bc} : m := rand \left( \{i \in 1..|O| \mid S_i = \min \left( \bigcup_{j \in 1..|I|} S_j \right) \} \right) \]

   - **Edge ee** \( g_{ee} : S_m := up(S_m) \).

   - **Edge be** \( g_{bc}: \exists i \in 1..|I| \quad T_i < \min \left( \bigcup_{j \in 1..|O|} S_j \right) \).

   \[ op_{bc} : m := rand \left( \{i \in 1..|I| \mid T_i = \min \left( \bigcup_{j \in 1..|O|} T_j \right) \} \right) \]
- Edge \( ee \) of TD: \( \neg \text{empty}(B_m) \), \( op_{ee} := \text{first}(B_m) \); \( B_m := \text{dequeue}(B_m) \), which updates \( T_m \) and \( B_m \) as explained in Sect. [III-B1a].

- Edges \( snd \) of TD: \( m = i \).

- Edges \( recv \) of TD: \( \neg \text{full}(B_i) \).

This TD enforces the rules given in Sect. [II] at a node level. For instance, \( g_{\text{out}} \) 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.

![Fig. 4. A node TD example (one input and two outputs)](image)

put 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, \( op_{be} \) 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.

**b) System:** A DSAAM system is the parallel composition\
\[
\{\text{Init}\} \parallel N_i \mid i \parallel N_i \mid \ldots \parallel N_i \mid \ldots
\]
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 \( E^S = \bigcup_{i \in I} E_i^S \) with \( E_i^S = \bigcup_{x \in \alpha^i} \text{snd}_x^i \).

Similarly, \( E^R = \bigcup_{i \in I} E_i^R \) with \( E_i^R = \bigcup_{j \in \beta^i \cap |O_i^j|} \text{recv}_j^i \). Now, the matching function \( M \) (Sect. [III-A3b]) is simply derived from the matching function \( F \): an edge \( \text{recv}_k^i \) belongs to the set of the matching edges of an edge \( \text{snd}_l^i \) iff \( IF_l^i \in F(OF^i_j) \).

**Shared variables:** \( \mathcal{U} = \{M_{\text{Msg}}\} \cup \{\alpha^1, ..., \alpha^x\} \) is the set of shared variables (Sect. [III-A3a]) where \( M_{\text{Msg}} \) is the message passing variable and each \( \alpha^i \) 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 \( \alpha^i := \text{F}(OF^i_m) \) is added to \( op_{be} \).

- On each edge \( snd^i \), the guard is conjuncted with the expression \( \alpha^i \neq \emptyset \) and the operation \( \text{Msg} := \text{up}(S_m) \) is added.

- On each edge \( recv^i \), the guard is conjuncted with the expression \( IF^k_i \in \alpha^i \) where \( i \) is the node identity of the only output that serves \( I^k \) (the only output \( O^j \) such that \( IF^k_i \in F(OF^i_j) \), Sect. [III-B1d] and Sect. [III-B1e]). Each \( recv^i \) edge is augmented with the operations \( B^j_k := \text{enqueue}(B^i_k, \text{Msg}) \).

- Edge \( ee^i \) is guarded with the expression \( \alpha^i = \emptyset \).

**Timestamps:** To start the system in a time-consistent state, we require enforcing the following in the TS of the DSAAM system: if \( q_0 \) is the initial state, then \( q_0(T^i_j) = q_0(S^k_i) \) iff \( IF^k_i \in F(OF^i_j) \).

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 \( \alpha \) 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 \( S = (N, F) \) such that:

- Outputs: \(|O^1| = 2 \) and \(|O^2| = |O^3| = 1\).
- Inputs: \(|I^3| = 3 \) and \(|I^2| = |I^1| = 1\).

- The flow function \( F: F(OF^i_j) = \{IF^2_j, IF^3_j\} \), \( F(OF^i_3) = \{IF^3_j\}, F(OF^i_2) = \{IF^2_j\}, F(OF^i_1) = \{IF^1_j\} \).

Using the rules given in Sect. [III-B2a] and Sect. [III-B2b] we derive the operational semantics of \( S \) 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 \( op_{1} \) is \( B_1^1 \)).

Let us now explain how this works through an emission scenario, by taking the transition \( t_{\text{emit}} \) (see mapping edges to transitions in Sect. [III-A3c]), and assuming the chosen output is \( O^1 \) \((m = 1\) after taking \( t_{\text{emit}} \)). In this case, the message should be sent to both inputs \( I^2_1 \) and \( I^3_1 \) \((\alpha^1 = \{IF^2_1, IF^3_1\}) \). Now, the enabled transitions in the system involving \( N_1 \) depend on the status of buffers \( B^2_1 \) and \( B^3_1 \). If none of them is full, one of the transition \( t_{\text{snd}^1_1, \text{recv}^3_1} \) or \( t_{\text{snd}^3_1, \text{recv}^2_1} \) is taken. In the former (resp. latter) case, \( M_{\text{Msg}} \) is emitted on input \( I^2_1 \) (resp. \( I^3_1 \)) and \( IF^3_1 \) (resp. \( IF^2_1 \)) is removed from \( \alpha^1 \). Subsequently, \( t_{\text{snd}^1_1, \text{recv}^3_1} \) (resp. \( t_{\text{snd}^3_1, \text{recv}^2_1} \)) is no longer enabled and the only possible transition involving \( N_1 \) is \( t_{\text{snd}^1_1, \text{recv}^3_1} \) (resp. \( t_{\text{snd}^3_1, \text{recv}^2_1} \)) because all other transitions involving \( N_1 \) are disabled (transitions involving \( snd^2_1 \) are disabled because \( m \neq 2 \) and \( t_{\text{emit}} \) is disabled because \( \alpha^1 \neq \emptyset \)). Consequently, the remaining input to serve is delivered \( M_{\text{Msg}} \) by taking \( t_{\text{snd}^1_1, \text{recv}^3_1} \) (resp. \( t_{\text{snd}^3_1, \text{recv}^2_1} \)) and \( \alpha^1 \) becomes empty, which enables ending the emission by taking \( t_{\text{emit}} \).

**C. Proof of progress**

**DSAM** 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:

\[
\forall q \in Q, \exists t \in \rightarrow : t(q) \neq \emptyset
\]  

(1)
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. 7).

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

\[ \exists q_d \in Q_r : \forall t \in \rightarrow_s : t(q_d) = \emptyset \]  
(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 I} S_j^i (resp. S = \bigcup_{i \in I} S^i). Similarly, T^i = \bigcup_{i \in I} T_j^i and T = \bigcup_{i \in I} T^i (for input timestamps).

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

\[ S^k = \min(S) \]  
(3)

Now, we know that, at q_d, the current vertex \pi_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 \} which results in a contradiction and thus the falseness of hypothesis 2, i.e., the truth of Theorem 1.

(a) q_d(\pi_k) \neq Wa^k: Let us assume q_d(\pi_k) = Wa^k. We see that q(g_{bc}) \lor q(g_{bc}) is a tautology (N is a well-ordered set) and thus either t_{bc}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) 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_{bc}) is false, that is:

\[ empty(B_{bc}^k) \]  
(4)

Let O_k^i be the output connected to I_m^k, that is IR_{bc}^k \in F(O_k^i). Since B_{bc}^k is empty, the valuation of T_m^k, holding the timestamp of the next message to consume on I_m^k, is lower-bounded by S^i_k (it is possible to dequeue up(S^i_k) (op_{ec}) before updating up(S^i_k) and up(S^i_k) \geq S^i_k).

\[ T_m^k \geq S^i_k \]  
(5)

From inequality 5 and inequality 3 we deduce:

\[ S^i_k \leq T_m^k \]  
(6)

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

\[ T_m^k < \min(S^i_k) \]  
(7)

The conjunction of 7 and 5 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) q_d(\pi_k) \neq Em^k: Let us assume q_d(\pi_k) = Em^k. We deduce that S^i_m = \min(S^i_k) and thus from 3:

\[ S^i_m = S^i_k \]  
(8)

From hypothesis 2, we deduce that t_{ec}^k(q_d) = t_{nd}^k(q_d) = \ldots = t_{nd}^k(\alpha) = \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:

\[ \alpha \neq 0 \land (\exists i, j \in I_x \mid IR_{bc}^i \in F(O_k^j) \land full(B_j^i)) \]  
(9)

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

\[ T_j^i < S^i_m \]  
(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^k), 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_j^i. This is a contradiction because T_j^i < S^i_m.
and \(S^k_p = \min(S)\) \((3)\). Therefore \(q_d(\pi_i) \neq Em^i\) (c’) We assume \(q_d(\pi_i) = Co^i\). From hypothesis \(2\) we deduce:

\[
\emptyset(B_{m^t}^i)
\]

From \(9\) and \(11\) we deduce \(m^t \neq j\), 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 \(O_y^z\) be the only output connected to the input \(I_{m^t}^i\) with buffer \(B_{m^t}^i\):

\[
I_{m^t}^i \in \mathcal{F}(OF_y^z)
\]

Because the buffer \(B_{m^t}^i\) is empty we have, following the same logic as to obtain \(5\)

\[
T_{m^t}^i \geq S_y^z
\]

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

\[
T_{m^t}^i \leq T_j^s
\]

Now, Combining \(14\), \(13\), \(10\)

\[
S_y^z \leq T_{m^t}^i \leq T_j^s \leq S_m^k
\]

Conjecturing \(3\), \(8\) and \(15\) is a fallacy, thus \(q_d(\pi_i) \neq Co^i\). Conjecturing \((a')\), \((b')\) 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, conjecturing \((a)\), \((b)\) 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 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 rise the interest of the formal community 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.

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. [6], [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 developer 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 $T$. 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 induce deadlocks. Future work includes an extended version of the DSAAM formal model with multiple flow types, and with automatic detection of such circuits.

References