Future Generation Computer Systems 56 (2016) 124–139

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Logic-based modeling of information transfer in cyber–physical multi-agent systems

Christian Kroiß [a,*], Tomáš Bureš [b]

[a] Ludwig Maximilian University of Munich, Institute for Informatics, Munich, Germany
[b] Charles University in Prague, Faculty of Mathematics and Physics, Prague, Czech Republic

HIGHLIGHTS

- Logic-based modeling of information transfer in multi-agent systems.
- Considers stochastic nature of communication in the cyber–physical setting.
- Allows property specification with first-order time-bounded LTL.
- Can be used for statistical model checking.

ARTICLE INFO

Article history:
Received 7 February 2015
Received in revised form
7 September 2015
Accepted 11 September 2015
Available online 25 September 2015

Keywords:
Statistical model checking
cyber–physical systems
Situation calculus
Discrete event simulation

ABSTRACT

In modeling multi-agent systems, the structure of their communication is typically one of the most important aspects, especially for systems that strive toward self-organization or collaborative adaptation. Traditionally, such structures have often been described using logic-based approaches as they provide a formal foundation for many verification methods. However, these formalisms are typically not well suited to reflect the stochastic nature of communication in the cyber–physical setting. In particular, their level of abstraction is either too high to provide sufficient accuracy or too low to be practicable in more complex models. Therefore, we propose an extension of the logic-based modeling language SALMA, which we have introduced recently, that provides adequate high-level constructs for communication and data propagation, explicitly taking into account stochastic delays and errors. In combination with SALMA's tool support for simulation and statistical model checking, this creates a pragmatic approach for verification and validation of cyber–physical multi-agent systems.

1. Introduction

With SALMA (Simulation and Analysis of Logic-Based Multi-Agent Systems) [1], we have recently introduced an approach for modeling and analysis of multi-agent systems that is aimed to provide a lightweight solution for approximated verification through *statistical model checking* [2] with the system model still being grounded on a rigorous formal foundation. SALMA's modeling language is based on the well-established *situation calculus* [3], a first-order logic language for describing dynamical systems.

In this paper, we provide an extension of SALMA (and the situation calculus in general) to explicitly address one aspect that is particularly important for *cyber–physical* [4] multi-agent systems, namely the distributed gathering and transfer of information. Agents not only have to continuously sense their environment, but also share these readings with other agents, acquire information of others, and participate in coordination activities. In the cyber–physical context, these information transfer processes are subject to stochastic effects, e.g. due to sensor errors or unreliable communication channels. Furthermore, accuracy and timing of information transfer processes can strongly influence the behavior of the whole system. In particular, the efficacy of mechanisms for self-adaptation or optimization typically degrades when certain time-constraints are violated or the accuracy of sensors is insufficient.

Using pure logical formalisms like the basic situation calculus for describing such scenarios results in rather verbose and low-level representations that are not practicable in more complex cases. What is needed instead are high-level constructs that establish a bridge between the underlying logical semantics and the typical requirements for modeling information transfer in

* Corresponding author.
E-mail addresses: kroiss@pst.ifi.lmu.de (C. Kroiß), bures@d3s.mff.cuni.cz (T. Bureš).

http://dx.doi.org/10.1016/j.future.2015.09.013
0167-739X/

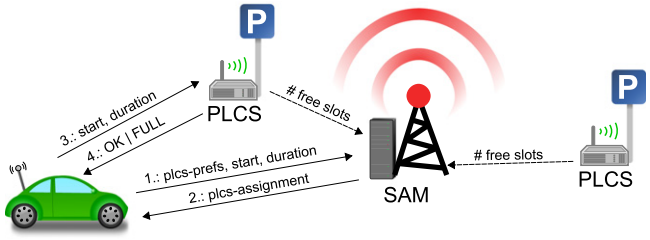

Fig. 1. Optimized parking lot assignment scenario.

multi-agent CPS. Although higher-level extensions on top of the situation calculus have been designed for related aspects like sensing and knowledge (e.g. [5]), there has, to our knowledge, not been a detailed reflection of information propagation in CPS in the context of the situation calculus.

We have therefore developed a generic model of information transfer that is based on a stochastic timed version of the situation calculus and allows capturing a wide range of effects that may be imposed on information transfer processes. Additionally, we have defined a set of macro-like abstractions for common information transfer scenarios within CPS, such as message passing or sensor data propagation. This creates a concise interface for the modeler that hides the stochastic details of information propagation but makes them fully accessible in simulation and verification.

In the following sections, we first set the picture by introducing an example from the e-mobility domain that will be used throughout the paper to demonstrate the developed concepts. We then shortly give some necessary background about the situation calculus that will be needed to understand the mechanisms described later. Then, in Section 4, we introduce the basics of the SALMA approach and formally define a core part of its simulation semantics in Section 5 as a foundation for the discussion of the information transfer model. The main contribution of this article starts in Section 6, where we introduce our generic model for information transfer and describe precisely its realization on the basis of the situation calculus. After that, in Section 7, we define several extensions to SALMA's modeling language that provide pragmatic abstractions for the information transfer model. This is continued in Section 8, where the focus is set on the use of SALMA for statistical model checking in the context of information transfer processes. As an evaluation of our approach, Section 9 discusses its application to the example that was introduced in the beginning. There, we show some experimental results and assess experiences regarding the benefits of our approach. Finally, we give an overview of related work before we end the paper with a short conclusion.

2. Example: optimized parking lot assignment

As a running example to illustrate our approach, we employ the e-mobility case-study of the ASCENS EU project[1] that has been described before, e.g., in [6]. The case study focuses on a scenario in which electric vehicles compete for parking lots with integrated charging stations (PLCS) in an urban area. The goal is to find an optimal assignment of PLCS to vehicles. Technically, the assignment is performed by an agent called super-autonomic manager (SAM) that coordinates a number of PLCS. The structure of the information transfer within this scenario is outlined in Fig. 1. The basic idea is that vehicles send *assignment requests* to the SAM, including a start time, a duration, and a list of preferred PLCS that is compiled by the vehicle's on-board computer. The SAM tries to find optimal suggestions for parking lot assignments, based on the knowledge about driver's intentions, and on occupancy information that is sent repeatedly by the PLCS.

True to the distributed CPS principle, all the agents (vehicles, PLCS, SAM) are autonomous and communicate via some wireless data transmission infrastructure like a VANET or 3G/4G network. This implies that neither transmission delays nor the possibility of errors can be neglected. However, timing clearly plays an important role in the scenario described above. First of all, the reservation service would simply not be accepted if the delay between reservation requests and reservation responses was too high. Also, the communication timing affects the convergence of the optimization, thus directly it influences the functionality of the distributed CPS.

3. Background: situation calculus

In this article, we show how to capture the timing aspects in situation calculus models, while maintaining a practical level of abstraction that focuses on the core "business-level" functionality. However, before doing so, we briefly summarize the main principles of the situation calculus in this section, so as to provide a necessary background for the rest of the paper.

The situation calculus [3] is a first-order logic language for modeling dynamic systems. Its foundation is based on the notion of *situations*, which can be seen as histories of the world resulting from performing *action sequences*. Actions can either be deliberately executed by agents or *exogenous*, i.e. external *events* caused by the environment. Situation terms are then formed recursively by the function $do(a, s)$ that denotes the execution of action a in situation s. Consequently, the term

$$do(a_n, do(a_{n-1}, do(\ldots, do(a_1, S_0)\ldots)))$$

stands for the situation that results when the action sequence $\langle a_1, \ldots, a_n\rangle$ is executed in the initial situation S_0.

The state of the world in a given situation is defined by the set of all *fluents*, which are situation-dependent predicates or functions. Since the models discussed here are meant to be used in *discrete event simulation*, time itself is simply modeled as an integer fluent named *time* that is increased with each simulation step. How other fluents are affected by *actions* and *events* is defined by *successor state axioms (SSAs)* that recursively relates the next situation to the current one. In fact, a situation calculus model has to contain one SSA for every fluent that define exactly when a *boolean (relational) fluent* is true, or when a *functional fluent* has a certain value. As a simple example, the following axiom states that a vehicle is driving in a situation $do(a, s)$ either when a start event occurred, or when no stop event occurred and the vehicle had been driving in the situation before.

$$driving(vehicle, do(a, s)) \equiv a = start(vehicle) \vee (\neg(a = stop(vehicle)) \wedge driving(vehicle, s)).$$

Additionally, a situation calculus model also contains *precondition axioms* that define whether or not an action or event is possible in a given situation. In general, both the effects of actions and events, and also the occurrence of exogenous actions, are of stochastic nature. Consequently, simulation involves sampling from a set of probability distributions that the modeler can define as part of the simulation's configuration (cf. Section 7).

In SALMA, the situation calculus is used together with a forward reasoning technique called *progression* [3, chap. 9] that basically uses the successor state axioms to create a new *snapshot* of the world state and uses this as the initial situation for the next simulation step. In contrast to the original situation calculus reasoning method, *regression* [3, chap. 4.5], progression actually "forgets the past" and is therefore not suited for many theorem

[1] www.ascens-ist.eu.

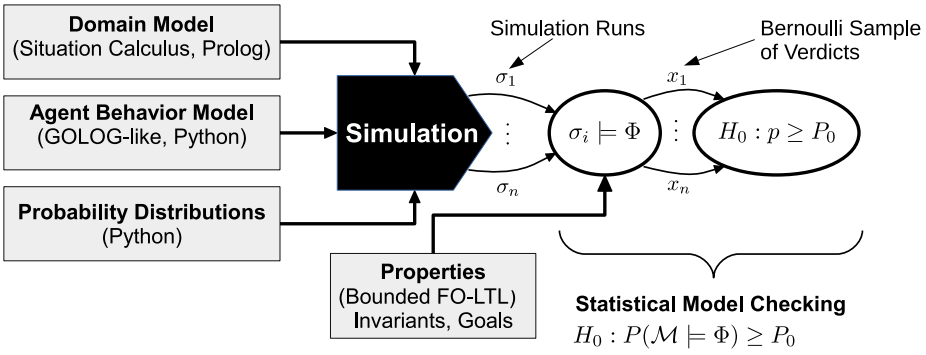

Fig. 2. Overview of the SALMA approach.

proving applications. However, since regression works on situation terms that encode the whole action sequence since the initial situation, it is not applicable in complex, long running simulations.

One of the most prominent applications of the situation calculus is GOLOG [7], a language that combines elements from procedural with logic programming. It has been used for modeling and implementation in various domains, ranging from robotics to the semantic web. In particular, GOLOG's core principles have strongly inspired the SALMA approach, which is introduced in the next section.

4. The SALMA approach

In [1], we have introduced SALMA (Simulation and Analysis of Logic-Based Multi-Agent Systems), an approach that adapts the concepts of the situation calculus and GOLOG for discrete event simulation and *statistical model checking*. In this section, we first give a broad overview of SALMA, and in the next section, we discuss the core semantics of its simulation engine in more detail. Altogether, this part is meant to provide the necessary background and context for the presentation of the information transfer model in the remainder of this article.

The SALMA approach is outlined in Fig. 2. The *domain* model, i.e. the general mechanisms of the simulated world, is described by means of situation calculus axioms that are encoded in Prolog. Based on this axiomatization, the modeler defines the behavior of agents by equipping them with one or multiple processes that can be defined using SALMA's procedural process definition language (SALMA-PDL). Realized as an internal domain specific language (DSL) [8] within Python, the SALMA-PDL offers the usual control flow constructs like loops and conditionals, but also provides means to access the underlying situation calculus model, in particular by performing actions, reading fluent values, and observing events. For example, the code fragment in Fig. 3 defines a process that reacts to the event *called* by iteratively performing the action *moveRight* (*Act*) and waits for the completion of each step, which is indicated by a *finishStep* event. In order to stop the movement at the target, the fluents *xpos* and *targetX* are accessed in the loop condition.

Processes can be set up with different scheduling schemes, namely one-time execution, periodic, or triggered as in Fig. 3, i.e. executed when a given condition evaluates to true. The ability to freely combine these options allows the modeler to realize various agent architectures, in particular reactive and layered architectures [9] that are widely used in robotics and other branches of cyber–physical systems.

With the models for the domain and the agents' behavior in place, a concrete *simulation experiment* is created by defining initial values and *probability distributions* for stochastic actions and events.

```
p1 = Procedure([
        While("xpos(self) < targetX(self)", [
            Act("move_right", [SELF]),
            Wait("occur('finish_step', self)")])]
mover = TriggeredProcess(p1, "occur(called(self))")
```

Fig. 3. Simple procedure in SALMA-PDL.

The simulation engine uses a simulation world view that can be seen as a combination of *event scheduling* and *activity scanning*, similar to the three-phase approach in discrete event simulation [10]. For *scheduled events*, the occurrence time is determined in advance by sampling from a (conditional) probability distribution. Additionally, for *immediate events*, the simulation engine decides for each time step separately whether the event should occur or not—similar to activity scanning. While the second approach obviously increases computational effort, it can be better suited for capturing highly dynamic effects, e.g. when the position of a moving agent has significant impact on the latency of an ongoing communication process.

In addition to the system model, a set of invariants and goals can be specified using SALMA's property specification language, which is mainly a first-order version of linear temporal logics (LTL) [11] with time-bounds for the temporal modalities. Since the simulated system model is also described by means of first-order logics, the property specification language is able to provide a very detailed and direct access to the system's state (i.e. fluents), actions, and events. For example, Fig. 4 shows a property that asserts that after any robot is called, it reaches its target within 100 time units while keeping a minimum distance of 50 to all other robots.

Given the system model together with invariants and goals, the SALMA interpreter performs *discrete event simulations*. For each simulation run, the engine eventually decides whether it satisfies the given properties or not. The set of resulting verdicts yields a *Bernoulli sample* that is used to test the statistical hypothesis $H_0 : p \geq P_0$ which asserts that the probability of a success (a run fulfills the property) is at least as high as a given lower bound. By using the sequential probability ratio test (SPRT) by A. Wald [12], the number of required simulation runs for given statistical error bounds can be determined dynamically. Additionally, any methods for interval estimation of binomial proportions can be used to estimate p directly (see [13]).

This way of approximative assertion of properties defined by temporal logics is generally called *statistical model checking* [2] and provides a pragmatic alternative to exact model checking techniques that does not suffer from the same scalability problems

```
forall(r1:robot, implies(occur(called(r1)),
    until(100, forall(r2:robot, implies(r2 \= r1, dist(r1, r2) >= 50)), atTarget(r1))))
```

Fig. 4. Example property defined with SALMA's property specification language.

since only individual simulation runs are inspected instead of the complete state space.

5. SALMA simulation semantics

This section formalizes the operational semantics of the core part of SALMA's simulation framework. The goal is to provide the necessary background for the following sections. Rather than to cover every detail. To save space, we intentionally omit details of the SALMA's semantics that are not relevant for the description of the model for information transfer as featured in the paper.

As a starting point, we define the notion of the simulated system as a combination of all declarations together with the world state.

Definition 1 (*Simulated System*)**.** A simulated system is defined by the following tuple:

$$Sys = \langle Decl, \mathcal{D}^{Dom}, Agents, (Procs_a)_{a \in Agents}, Prob, S_0, St \rangle.$$

Here, *Decl* is the set of all declaration statements for sort, sort hierarchies, fluents, (exogenous) actions, connectors, and ensembles. In conjunction with this, $\mathcal{D}^{Dom}$ denotes the *basic action theory* [3] that forms the situation calculus basis for the simulation, i.e. the complete set of axioms that define how the system can progress in response to actions and events. Furthermore, $(Procs_a)_{a \in Agents}$ is an indexed family of process definitions that define the agents' behavior, and *Prob* stands for the set of probability distributions that are used by the simulation to schedule events and to choose probabilistic action outcomes. Finally, S_0 is the *initial situation*, i.e. the set of values for fluent instances that is used at the start of the simulation run. ■

The most important part of the definition above for describing the simulation semantics, is obviously the system state, which is by itself a combination of several structures that are manipulated during simulation.

Definition 2 (*System State*)**.** The current state of a simulated system is given by the following tuple:

$$St = \langle P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S \rangle.$$

Here, P_{run}, P_{act}, P_{wait}, and P_{idle} are sets of *process states* (see below) that describe the processes that are currently being executed, performing actions, waiting, or idle, respectively. *Act* is the set of pending actions that are yet to be executed in the current simulation step, Evt is the event schedule, and S represents the current situation, i.e. the current set of values for all fluent instances of the system. ■

The process state descriptions mentioned above combine all information about the current state of each process. As mentioned in the beginning of this section, the details about procedure execution are not presented here to simplify the description.

Definition 3 (*Process State*)**.** A process state is defined by a tuple of the following form:

$$(pid, a, n_{cur} \circ \sigma, \eta, \pi).$$

Here, *pid* is a process identifier, a is the agent that executes the process and n_{cur} is the *process control node* that is executed next. The suffix of the process, i.e. the remaining control nodes of the process body that will be executed after n_{cur}, is denoted by σ, and the operator $\circ$ is used to represent sequence composition. Finally, η is the process evaluation context that defines the mappings of variables to values, and π is the procedure call stack that stores the environment that will be restored when the current procedure is exited. ■

At the *initial state* of a simulation, the *initial situation* S_0 holds, i.e. all fluent instances are set to their initial values. Additionally, all processes are in the *idle* state and neither actions nor events are scheduled.

Definition 4 (*Initial System State*)**.** Let S_0 be the initial situation of the system in the sense of the situation calculus. Furthermore, let $PBody_p$ be the full control node sequence of the procedure declaration of process p. Then the initial state of the simulation is given by

$$St_0 = \langle P^0_{run}, P^0_{act}, P^0_{wait}, P^0_{idle}, Act_0, Evt_0, S_0 \rangle$$

where

$$P^0_{run} = P^0_{act} = P^0_{wait} = Act_0 = Evt_0 = \emptyset$$
$$P^0_{idle} = \{(pid, a, PBody, \emptyset, \emptyset) \mid a \in Agents \wedge (pid, PBody) \in Procs_a\}.$$ ■

Concrete control nodes, that can appear in n_{cur} or σ from above, include the usual control flow statements like conditionals or loops. Aside from that, there is the *Act* statement with which an agent can execute *actions*. In fact, this is the only option for an agent to influence its environment — namely through the effect of the executed actions in the sense of the situation calculus. However, the progression is not performed directly for each *Act* call. Instead, the current interpretation of the action term is added to the set of pending actions. At the same time, the process is suspended temporarily until the action has been handled.

Definition 5 (*Action Execution*)**.** Let α be an action term that possibly contains variables. Then,

$$\langle \{(pid, a_s, Act(\alpha) \circ \sigma, \eta, \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S \rangle$$
$$\longrightarrow \langle P_{run}, P_{act} \cup \{(pid, a_s, \sigma, \eta, \pi)\}, P_{wait}, P_{idle}, Act \cup \{[\![\alpha]\!]_{S,\eta}\}, Evt, S \rangle.$$ ■

Concordant with the postponed execution mentioned in the last definition, the system performs *progression* steps only when all active processes are currently blocked, i.e. they are either waiting or performing actions. In this case, both pending actions and events that are due for the current time step are performed in random order.

Definition 6 (*System Progression*)**.** Let α and ϵ be ground terms that denote a valid concrete action or event, respectively. Furthermore, let $t = time(S)$ be the current time and $Progress(\alpha, S)$ the world state that results from performing a progression step for action α in the current situation S (see [3, chap. 9]). Then,

(a) *Actions*:

$$\langle \emptyset, P_{act}, P_{wait}, P_{idle}, \{\alpha\} \cup Act, Evt, S\rangle \longrightarrow \langle \emptyset, P_{act}, P_{wait}, P_{idle}, Act, Evt, S'\rangle$$
$$\text{where } S' = \begin{cases} Progress(\alpha, S) & \text{if } Poss(\alpha, S) \\ S & \text{if } \neg Poss(\alpha, S). \end{cases}$$

(b) *Events*:

$$\langle \emptyset, P_{act}, P_{wait}, P_{idle}, Act, \{(\epsilon, t)\} \cup Evt, S\rangle \longrightarrow \langle \emptyset, P_{act}, P_{wait}, P_{idle}, Act, Evt, S'\rangle$$
$$\text{where } S' = \begin{cases} Progress(\alpha, S) & \text{if } Poss(\epsilon, S) \\ S & \text{if } \neg Poss(\epsilon, S). \end{cases} \blacksquare$$

After all currently scheduled agent actions have been performed, blocked processes are reactivated. This first includes all processes that are currently executing actions. Additionally, waiting and idle processes can be scheduled in this phase when the corresponding conditions are satisfied.

Definition 7 (*Process Reactivation*)**.** Let $cond(p)$ denote a condition on which the process p is waiting after executing a *Wait* statement, and let $trigger(p)$ be a condition that, if true, triggers an idle process to be executed in the current step. Then,

$$\langle \emptyset, P_{act}, P_{wait}, P_{idle}, \emptyset, Evt, S\rangle \longrightarrow \langle P_{run}, \emptyset, P'_{wait}, P'_{idle}, \emptyset, Evt, S\rangle$$
$$\begin{aligned} \text{where } P_w^+ &= \{p_w \mid p_w \in P_{wait} \wedge [\![cond(p_w)]\!]_S = \top\} \\ P_i^+ &= \{p_i \mid p_i \in P_{idle} \wedge [\![trigger(p_i)]\!]_S = \top\} \\ P_{run} &= P_{act} \cup P_w^+ \cup P_i^+ \\ P'_{wait} &= P_{wait} \backslash P_w^+ \\ P'_{idle} &= P_{idle} \backslash P_i^+. \end{aligned}$$ ■

An event can be added to the event schedule either (a) *instantaneously*, i.e. at the current moment if it is found to be possible and chosen by "coin flipping", or (b) *anticipatory*, i.e. at a time point in the future that is chosen according to a specific probability distribution. In general, scheduling is performed iteratively until no further events can be scheduled, although every concrete event can only be scheduled once. Additionally, there may be cases where there can be only one of a set of several events—i.e. they form an *event choice*. An example for such a case will be presented later in Section 6 with the two events *transferStarts* and *transferFails*.

Definition 8 (*Event Scheduling*)**.** Let ϵ be an event term, ΔT a random variable that models a delay, and $Occur_\epsilon$ a random variable that models whether an event should occur in the current simulation step ($Occur_\epsilon = 1$) or not ($Occur_\epsilon = 0$). For both $Occur_\epsilon$ and ΔT, it is assumed that appropriate probability distributions have been chosen by the modeler. Furthermore, the predicate $Sched(\epsilon, S)$ is used to determine whether event ϵ is *schedulable* in the current situation S. Whether two events are mutually exclusive as part of an *event choice* is determined by the predicate $exclusive(\epsilon_1, \epsilon_2)$. Finally, the notation $\xrightarrow{p}$ is used to express that the given state transition occurs with probability p.

Given the definitions above, assume that Φ_0 and one of the two conditions Φ_1 or Φ_2 is satisfied:

$$\begin{aligned} \Phi_0 &\equiv (\nexists t'.\, (\epsilon, t') \in Evt) \\ &\quad \wedge (\nexists t'' \nexists \epsilon'.\, exclusive(\epsilon, \epsilon') \wedge (\epsilon', t'') \in Evt) \\ \Phi_1 &\equiv type(\epsilon) = immediate \wedge Poss(\epsilon, S) \\ \Phi_2 &\equiv type(\epsilon) = scheduled \wedge Sched(\epsilon, S). \end{aligned}$$

Then, for any value $\delta t \in \mathbb{N}_0$, the event instance ϵ is scheduled δt time units after the current time step with probability p, i.e.

$$\langle \emptyset, P_{act}, P_{wait}, P_{idle}, Act, Evt, S\rangle \xrightarrow{p} \langle \emptyset, P_{act}, P_{wait}, P_{idle}, Act, \{(\epsilon, [\![time]\!]_S + \delta t)\} \cup Evt, S\rangle$$

where

$$p = \begin{cases} Pr(Occur_\epsilon = 1 | \epsilon, S) & \text{if } \Phi_1 \text{ and } \delta t = 0 \\ Pr(\Delta T = \delta t | \epsilon, S) & \text{if } \Phi_2 \\ 0 & \text{otherwise.} \end{cases} \blacksquare$$

When no agent processes can progress further and no more actions or events can be executed in the current simulation step, the system progresses to the next time point that is relevant either (a) because an event is scheduled, (b) because a periodic process is scheduled to be executed, or (c) because an event will become possible or schedulable at this time.

Definition 9 (*Time Advance*)**.** Let the predicate *EligibleEvent* be defined so that it is true when an event instance can either happen immediately (in the current time step) or can be scheduled, i.e.

$$\begin{aligned} EligibleEvent(\epsilon) \equiv\ & (type(\epsilon) = immediate \wedge Poss(\epsilon, S)) \\ & \vee (type(\epsilon) = scheduled \wedge Sched(\epsilon, S)). \end{aligned}$$

Furthermore, the predicate *StepEnd* tests whether the system has arrived in a state where no further events can be scheduled and no process can be activated:

$$\begin{aligned} StepEnd \equiv\ & (\nexists p_w \in P_{wait}.\, [\![cond(p_w)]\!]_S = \top) \\ & \wedge (\nexists p_i \in P_{idle}.\, [\![trigger(p_i)]\!]_S = \top) \\ & \wedge \nexists \epsilon.\, EligibleEvent(\epsilon, S). \end{aligned}$$

If *StepEnd* holds in the current system state, then

$$\langle \emptyset, \emptyset, P_{wait}, P_{idle}, \emptyset, Evt, S\rangle \longrightarrow \langle \emptyset, \emptyset, P_{wait}, P_{idle}, \emptyset, Evt, S'\rangle$$
$$\begin{aligned} \text{where } S' &= S[time \mapsto t_{next}] \\ t_{ev} &= min\{t \mid (\epsilon, t) \in Evt\} \\ t_{period} &= min\{t \mid \exists p \in P_{idle}.\, t = nextScheduleTime(p)\} \\ t_{scan} &= min\{t \mid \exists S, \epsilon.\, EligibleEvent(\epsilon, S) \wedge t = time(S)\} \\ t_{next} &= min\{t_{ev}, t_{period}, t_{scan}\} \end{aligned}$$

Here, $nextScheduleTime(p)$ denotes the time at which the next execution of the periodic process p is scheduled. ■

The definitions above together have two important implications:

Remark 1 (*Actions and Events do not have A Duration*)**.** The system does not enforce a limit for the number of actions and events performed at the same time step. This means that the modeler is responsible for assuring that the simulation does not exhibit Zeno-behavior (see [14]). Every activity that blocks a process for a certain duration has to be modeled by an action that is followed by a *Wait*-statement (e.g. in Fig. 3).

Remark 2 (*The Definitions Imply Weak Fairness Among Processes*)**.** Each time a process executes an action, it is only performed after all other processes had the chance to schedule an action execution. *All* actions from the action schedule *Act* are then performed in random order before any process can continue its execution.

6. A generic situation calculus model for information transfer

In order to use SALMA for analyzing scenarios like the one described in Section 2, concepts like sensing and communication have to be mapped to SALMA's modeling language framework. As a first step, we propose a generic model for information transfer in the situation calculus. This model is able to describe both sensing and inter-agent communication in a unified way and allows capturing stochastic effects with a variable level of detail.

In general, our approach is based on the notion that information is transferred from a *source fluent* to a *destination fluent* that is directly accessible by the receiving agent. The source fluent can

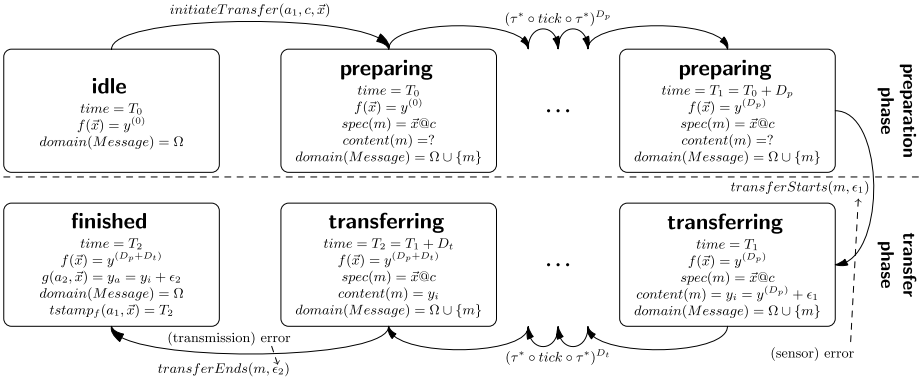

Fig. 5. Overview of the general information transfer model.

either represent a *feature of the physical world* or data created by some artificial process, e.g. a message queue. A *connector* is an entity that defines modalities of an information transfer process, including the fluent endpoints and the types and roles of participating agents. The *messages* that are transmitted over connectors are treated as first-level model citizens by representing them as entities of the dedicated sort *Message*. Both the content and the state of each message are stored separately by a set of fluents and evolve independently as result to several types of events. This representation provides great flexibility for the realization of arbitrary propagation structures.

In the following, the information model will first be introduced from a higher level as structured into two general phases, before Section 6.2 describes the information transfer paradigms supported by SALMA more thoroughly.

6.1. Information transfer phases

Based on the foundational concepts described above, we distinguish two phases of information transfer that are sketched in Fig. 5:

(A) The **preparation phase** starts when an agent (a_1 in Fig. 5) initiates the information transfer (sensing or communication). For that, he specifies a *connector* (c) and a parameter vector ($\vec{x}$) that fully qualifies the information source and, in case of a point-to-point transmission, contains the identity of the receiving agent. In response to this action, a new message (m) is created and initialized with the transfer metadata but without content yet. Depending on the concrete scenario, there can be various reasons for the actual transfer being delayed, e.g. initialization of sensors or communication devices. This means that there may be an arbitrary sequence of time steps (*tick* events) interleaved with actions and events (denoted as τ in Fig. 5) that may change the information source (f) but are not recognized by the agent. After that sequence, the actual value that is eventually used as message content can deviate from the information source value present at the time when the transfer was initiated.

(B) The **transfer phase** follows the preparation phase and begins when a *transferStarts* event occurs. At that point, the current value of the source fluent f is fixated as the content of the message that is now actually transferred to its destination over the connector c whose stochastic characteristics are specified within the simulation model. Like above, this phase may take an arbitrary amount of time during which unrecognized or unrelated actions and events occur. Eventually, a *transferEnds* event finishes the transfer process. Thereupon, the destination fluent instance $g(a_2, \vec{x})$ is updated and the message entity is removed. This moment, as well as the starting points of both phases, are memorized in time-stamp fluents that can, for instance, be used to reason about the age of a measurement.

The diagram in Fig. 5 omits the fact that, due to malfunctions and disturbances in the environment, the transfer could *fail* at any time, which would be represented by an additional event *transferFails*. Additionally, the transfer process may be affected by stochastic errors that eventually cause the received value to deviate from the original input, which is reflected by the error terms ϵ_1 and ϵ_2 in the events *transferStarts* and *transferEnds*.

In general, both stochastic errors and delays are governed by a set of probability distributions that are used during simulation to decide when the events mentioned above occur and which errors they introduce. By adjusting these parameters, a wide variety of different scenarios can be modeled, ranging from nearly perfect local sensing to wireless low-energy communication with interferences. This topic is examined further in Section 6.4.

6.2. Information transfer paradigms

The generic abstract model presented above contains several degrees of freedom regarding, in particularly with respect to how the content of messages is set, and how the recipients are determined. These variation points have to be handled according to the particular kind of information transfer. In the following, we present four classes of information transfer that cover the typical scenarios in cyber–physical multi agent systems. To make this section more comprehensible, we present the situation calculus semantics of each class in a reduced form by means of annotated state diagrams. For a detailed axiomatization, we refer the reader to Appendix.

Generally, we distinguish the following basic paradigms:

1. **Channel-based communication:** An agent actively sends data to one or several other agents. The well-known channel paradigm fits well to the asynchronous communication style predominant in CPS and to the relational way of identifying information in the situation calculus. As usual, a distinction is made between the following communication schemes:
   (a) **Unicast (point-to-point)**: Fig. 6 shows a unicast message transfer from agent *src* to agent *dest* via channel c. Here, a message is sent from the source to a single destination that is specified when the transfer is initiated. In the situation calculus model, this means that the specification of the message $spec(m)$ is set when the message is created. This specification includes all relevant static information about the message, namely the channel (c), the message type (*uc* for *unicast*), source and destination, as well as source and destination roles (r_s and r_d, respectively) and the original message content (*msg*). The actual *transferred content* (fluent C_{trans}) of this message is set when the transfer starts, possibly tampered by an error ε originating in the preparation phase. When the message transfer ends successfully

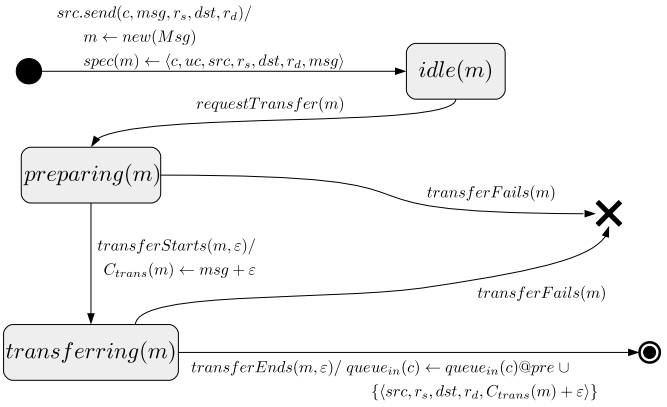

Fig. 6. Unicast channel based communication.

(event *transferEnds* occurs), a tuple is added to the channel's incoming message queue ($queue_{in}$). This message tuple identifies source and destination, their roles with respect to the channel definition, and the eventually received message content that again might incorporate an error originating from the transfer phase. In contrast, when the transfer fails (event *transferEnds* occurs) either during the preparation or transfer phase, the information is lost. In both cases, the message entity is removed from the model, i.e. from the domain of the message sort.

(b) **Multicast**: Here, the destination of the message transfer is not specified directly when the message is sent but instead via some shared address property. This case is shown in Fig. 7. Other than in the unicast case, the specification of the source message *m* does not contain the destination. Instead, the destinations are chosen on the arrival of a *transferStarts* event by evaluating the channels *ensemble predicate* (see Section 6.3). For each selected recipient, the source message is replicated and hence transferred on an independent path. In particular, terminating messages (*transferEnds* and *transfer*) occur independently for each message, which allows capturing phenomena that are caused by lack of synchronization or deviating information among agents.

2. **Generalized sensing:** an agent acquires information about a feature of the world that can be assessed through sensing. We further distinguish two types of sensing:

(a) **Direct (local) sensing**: Here the querying agent can produce the desired result on its own, although the sensing process may take a considerable amount of time and can be disturbed by internal or external factors. In Fig. 8, agent *ag* uses its local sensor *sen* to make a measurement. Different from the communication models explained above, the transferred content of the sensor message (Sen_{trans}) is not at all specified at the creation of the message but retrieved from the *sensed fluent* (f_{sen}) on arrival of *transferStarts*. Also, the received sensor values are not written to a queue like messages but are used to update a *local sensor view* fluent (*sen* in Fig. 8) that the agent can access.

(b) **Remote sensing**: Here the agent cannot observe the desired information itself by using its local sensors but has to gather information from one or several other agents. The remote sensing abstraction reflects the delays and disturbances of the involved communication processes but abstracts away their technical details. In particular, both for direct and remote sensing, the most recently retrieved value is made available in a *local fluent* that can be accessed in axioms and agent processes without considering the underlying infrastructure. In SALMA's situation calculus model, remote sensing is realized very similar to regular intentional multicast

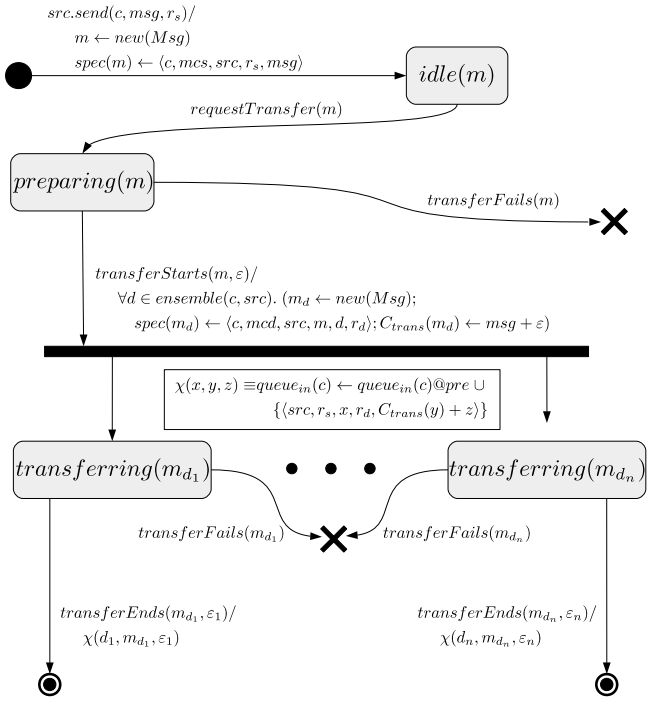

Fig. 7. Multicast channel based communication.

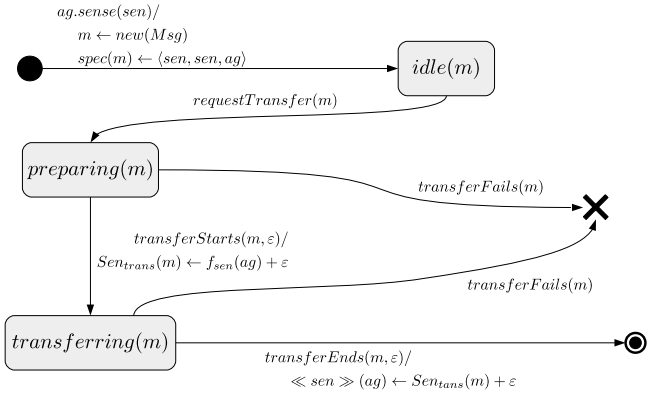

Fig. 8. Local sensing.

message communication as described above. Fig. 9 shows the transmission of data from agent *src*'s local sensor *sen* to recipients within the ensemble that is defined for the *remote sensor rsen*. The first main difference to multicast message transmission is that the transferred content is not defined explicitly by the sending agent but instead taken from the local sensor (*sen*) that is declared as the information source. Furthermore, the *ensemble* is specified from the perspective of the receiving agents. After the destination messages have been created, the rest of the transfer is identical to the multicast case from Fig. 7.

As mentioned above, remote sensing tries to hide the communication infrastructure and therefore present the acquired data similar to local sensor data. This is achieved with the *remote sensor view*, a fluent that keeps the most recent value from each source. The update step is shown if Fig. 10. It can be seen that the update step has to be performed actively by the receiving agents. This is intended as it resembles the distributed nature of remote sensing and allows capturing effects caused by delayed updates, etc. However, the update step is by default hidden from the

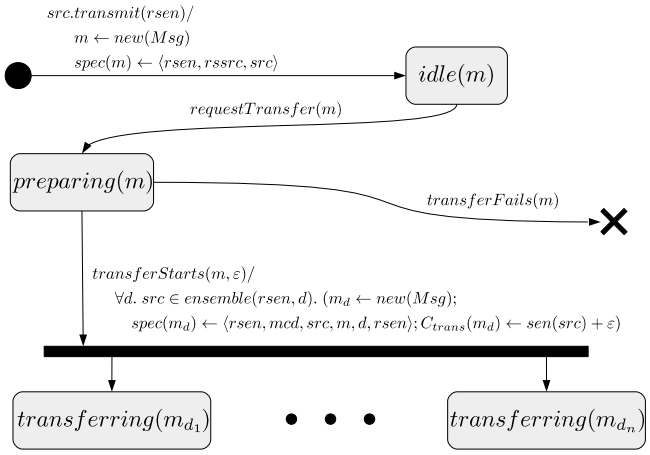

Fig. 9. Transmission of remote sensor data.

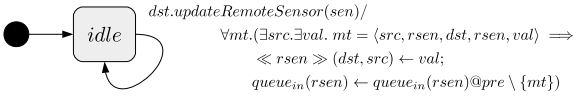

Fig. 10. Reception of remote sensor data.

```
ensemble(assignment, Vehicle, SAM, S) :-
  distance(Vehicle, SAM, D, S), D < max_comm_dist.
```

Fig. 11. Ensemble predicate example.

modeler as a background process that is installed automatically according to the remote sensor declarations in the model (see Section 7.3).

6.3. Predicate-based addressing

An important concern that arises in modeling multi-agent information propagation is how the set of receiving agents is determined. In many cases, it is either impossible or impracticable to do this statically. A particularly elegant alternative, supported by SALMA, is *predicate-based addressing* [15]. In this approach, the set of recipients for each information transfer is determined by a *characteristic ensemble predicate* that is evaluated for each (properly typed) agent pair. An *ensemble predicate* may describe *intentional selection criteria* as well as *intrinsic constraints* imposed by agent attributes or the environment. For instance, the Prolog code in Fig. 11, which is taken from the e-mobility example, declares an ensemble for the multicast channel *assignment*. The predicate selects all super-autonomous manager agents within a given maximum distance from a vehicle as recipients of messages that this vehicle sends on the channel.

6.4. Influence of the choice of probability distributions

Although the core information transfer mechanisms of SALMA are to a large extend determined by situation calculus axioms, the concrete characteristics of the modeled communication processes are governed by the choice of probability distributions for the occurrence of the events *transferStarts*, *transferEnds*, and *transferFails*. The SALMA framework provides access to common predefined distributions like Gaussian or Exponential, but also allows using custom distributions that can access all fluents within the system to derive parameters or make decisions. A common use case for distributions with situation-based parameters are message delays, i.e. transmission durations. For example, a reasonable choice for the duration of the transmission of messages with variable sizes would be a Gaussian distribution with a mean that is derived from the package size and the currently available channel bandwidth. If we denote with $chan(m)$ and $size(m)$ the channel and the size of message m, respectively, and if we assume there is a function $currentBandwidth(c, S)$ that returns the currently available bandwidth capacity of channel c at situation S, then a distribution for the delay of the *transferEnds* event could be given as in (6.1).

$$P(\Delta T = \delta t \mid transferEnds(m), S) \sim \mathcal{N}(\mu, \sigma^2)$$
$$\text{where } \mu = \frac{size(m)}{currentBandwidth(chan(m), S)}$$
$$\text{and } \sigma = F\mu. \tag{6.1}$$

The factor F above relates the standard deviation to the mean. This could be a constant value based on experience or also a function of factors like the current message traffic, etc.

The chosen value of δt is used to schedule the *transferEnds* event according to Definition 8 in Section 5. Before that, however, a choice is made between *transferEnds* and *transferFails* according to another distribution that represents the likelihood of failures. Additionally, the error amount ε that is introduced by *transferEnds* is sampled from a third distribution. As the preparation phase is treated analogically, the whole information transfer of a message is technically governed by six probability distributions altogether. This offers a huge variety of configuration choices and allows modeling a wide range of scenarios. While setting up the distributions for a simulation from scratch can clearly become a devious task for more complex models, the SALMA framework already offers several abstractions and pattern-based solutions that can reduce the effort significantly. In the future, these may be extended with additional pattern-based configuration macros that encode domain knowledge, experience, or empirical data.

7. Modeling abstractions for information transfer processes

To turn the generic information transfer model to a practical solution for modeling real-size systems, SALMA provides high-level constructs that reflect the way a modeler normally thinks about information transfer processes in a CPS. These constructs can be seen as macros that are internally mapped to situation calculus axioms, agent process fragments, and probability distributions. Altogether, this creates an instantiation of the generic schema from Section 6 that integrates seamlessly with the rest of the model. In the remainder of this section, the most important elements of this high-level language are introduced, and their integration into the general simulation semantics of SALMA and the information transfer model from Section 6 are explained.

7.1. Connector declaration macros

SALMA's high-level language support for communication and generalized sensing spans across several sections of the model. First, all *connectors*, i.e. local sensors, remote sensors, and channels are declared in the domain model. This is done with the Prolog predicates `channel`, `sensor`, and `remoteSensor`, that are shown in Fig. 12.

For `channel`, the modeler specifies a name for the channel, two *roles* with associated agent sorts, and the channel's *mode*, i.e. whether it is a singlecast or a multicast channel. All channels

```
channel(«name», «role1»:«sort1», «role2»:«sort2», «mode»).
sensor(«name», «ownerSort», «srcFluent»).
remoteSensor(«name», «ownerSort», «localSensor», «localSensorOwnerSort»).
```

Fig. 12. Connector declaration predicates.

```
channel(assignment, veh:vehicle, sam:plcssam, unicast).
channel(reservation, veh:vehicle, plcs:plcs, unicast).
sensor(freeSlotsL, plcs, freeSlots).
remoteSensor(freeSlotsR, sam, freeSlotsL, plcs).
```

Fig. 13. Connector declarations in the e-mobility example.

are bi-directional and the roles are used to distinguish message queues.

The `sensor` declaration defines the name of the local sensor, the sort of agents that own this sensor, and the fluent that represents the actual information source. This fluent is supposed to be qualified solely by the owning agent, i.e. it must be a function of the form $Agent \times Situation \rightarrow T$ with T being an arbitrary type. Similarly, the `remoteSensor` declaration establishes a sensor at the owner but connects it not to a fluent but to local sensors that are owned by other agents. Both local and remote sensor declarations add fluents of the same name to the model that provide a current view on the acquired information (see Section 6.2). Additionally, a time-stamp fluent is installed for each sensor that records the time of the latest measurement or remote data retrieval, respectively. Finally, the declarations for both local and remote sensors are used to automatically install background processes that hide the sensing infrastructure (see Section 7.3).

An example for the use of the predicates described above can be found in Fig. 13 that contains all connector declarations from the e-mobility example.

Here, `assignment` is defined to be a channel over which agents of the sort `vehicle` can communicate directly with agents of the sort `plcssam` in order to request and receive a PLCS assignment. The other channel `reservation` is used by vehicles to request slot reservations from PLCS agents and by the latter to acknowledge or deny these requests. The sensors of type `freeSlotsL` allow PLCS agents to count the current number of free slots at their station, i.e. access the fluent `freeSlots`. This information is propagated to the SAM via *remote sensors* of type `freeSlotsR` that effectively install channels and periodic background processes at each SAM and PLCS agent which transmit and receive the content of `freeSlotsL`, respectively.

7.2. Specialized process elements for information transfer

With the necessary declarations in place, the communication and sensing infrastructure can be used in agent processes by means of several special statements of the SALMA process definition language. As an example, the process `p1` shown in Fig. 14 is installed on PLCSSAM agents in the e-mobility example. It handles incoming requests from vehicles, calculates optimal assignments, and sends them back to the vehicles.

The process `p1` is executed when messages are available at the SAM's incoming message queue of the `assignment` channel. First, all available assignment requests are retrieved from the queue with a call to `Receive` which stores a message list in the variable `req`. The actual assignment selection logic is integrated by means

```
pprocreq = Procedure([
    Receive("assignment", "sam", "assignment_requests"),
    Assign(assignments, processRequests),
    Iterate(assignments, [v, p],
        Send("assignment", Term("aresp", p), "sam", v, "veh"))])
p1 = TriggeredProcess(pprocreq,
        "message_available", [SELF, "assignment", "sam"])
```

Fig. 14. Assignment request processing procedure in e-mobility example.

of an external Python function `processRequests`, which is not presented here for brevity's sake. Through the `Assign` statement, the function is called with the received request list as a parameter and the function's result is stored in the variable `assignments`. One of the most important inputs for this optimization is the number of free slots at each PLCS. This information is made available by the remote sensor `freeSlotsR` from above that transparently gathers occupancy information from all PLCS (see Section 7.3). The result of `processRequests`, stored in `assignments`, is a list of tuples that assign each requesting vehicle to a PLCS. The agent process iterates over this list and sends to each vehicle `v` in that list a response term that contains the PLCS id `p`.

Based on the definitions from Sections 5 and 6, the process elements used above can be defined accurately within the context of SALMA's simulation semantics and the situation calculus for information transfer. The most basic case is sending messages on a unicast channel like `assignment` from the example above.

Definition 10 (*Unicast Send*)**.** Let a_s be an agent and m a fresh message. Also, let c, msg, $dest$, r_s, and r_d be terms that can be evaluated at the current simulation state to a unicast channel, a viable message term, an agent, and roles defined for channel c, respectively. Then

$$\begin{aligned}
&\langle\{(a_s, \mathbf{Send}(\mathbf{c}, \mathbf{msg}, \mathbf{r_s}, \mathbf{dst}, \mathbf{r_d}) \circ \sigma, \eta, \pi)\} \cup P_{run}, P_{act}, P_{wait}, \\
&\quad P_{idle}, Act, Evt, S\rangle \\
&\longrightarrow \langle\{(a_s, Act(requestTransfer(m)) \circ \sigma, \eta, \pi)\} \\
&\quad \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S'\rangle \\
&\longrightarrow^* \langle\{(a_s, \sigma, \eta, \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S''\rangle
\end{aligned}$$

where $S' = S[domain(Msg) \mapsto [\![domain(Msg)]\!]_S \cup \{m\}, spec(m) \mapsto \langle [\![c]\!]_{S,\eta}, uc, a_s, [\![r_s]\!]_{S,\eta}, [\![dst]\!]_{S,\eta}, [\![r_s]\!]_{S,\eta}, [\![msg]\!]_{S,\eta}\rangle]$ and $S'' = S'[awaitingTransfer(m) \mapsto \top]$. ■

As described in Section 6.2, the model for multicast channel-based communication differs from the unicast case mostly in the transmission phase. In fact, sending a multicast message merely means to leave out the destination.

Definition 11 (*Multicast Send*)**.** Let a_s, m, msg, and r_s be defined as in Definition 10. However, let c now refer to a *multicast channel*.

Then

$$\langle \{(a_s, \mathbf{Send}(\mathbf{c}, \mathbf{msg}, \mathbf{r_s}) \circ \sigma, \eta, \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S\rangle \longrightarrow \langle \{(a_s, Act(requestTransfer(m)) \circ \sigma, \eta, \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S'\rangle \longrightarrow^* \langle \{(a_s, \sigma, \eta, \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S''\rangle$$

where $S' = S[domain(Msg) \mapsto [\![domain(Msg)]\!]_S \cup \{m\}, spec(m) \mapsto \langle [\![c]\!]_{S,\eta}, mcs, a_s, [\![r_s]\!]_{S,\eta}, [\![msg]\!]_{S,\eta}\rangle]$

and $S'' = S'[awaitingTransfer(m) \mapsto \top]$. ■

Agents that are receiving messages from unicast or multicast channels must select those messages from the channels message queue that are sent to the receiving agent and that match the requested destination role. In fact, the `Receive` statement removes all those messages from the queue and stores the resulting set in a variable.

Definition 12 (*Receive*)**.** Let a, refer to an agent, and let c be a term that denotes a channel in the current evaluation context η. Furthermore, let the evaluation of r_d in η refer to a role of c. Finally, let v be a variable name that is unbound in η. Then,

$$\langle \{(a, \mathbf{Receive}(\mathbf{c}, \mathbf{r_d}, \mathbf{v}) \circ \sigma, \eta, \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S\rangle \longrightarrow \langle \{(a_s, Act(cleanQueue(a_s, c, r_s)) \circ \sigma, \eta', \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S\rangle \longrightarrow^* \langle \{(a_s, \sigma, \eta', \pi)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, S'\rangle$$

where $\mathcal{M} = \{mt \mid mt \in [\![queue_{in}(c)]\!]_{S,\eta} \wedge \exists a_s \exists r_s \exists \vartheta.\, mt = \langle a_s, r_s, a, [\![r_d]\!]_{S,\eta}, \vartheta\rangle\}$

$\eta' = \eta[v \mapsto \mathcal{M}]$

$S' = S[queue_{in}([\![c]\!]_{S,\eta}) \mapsto [\![queue_{in}(c)]\!]_{S,\eta} \setminus \mathcal{M}]$. ■

The three statements presented above are actively used within agent processes to realize the respective agent's role in message-based communication processes. Additionally, there are three further statements that are necessary to implement the information transfer processes described in Section 6.2:

- `Sense(c)` initiates local sensing on sensor `c` as shown in Fig. 8.
- `TransmitRemoteSensorReading(rs)` initiates the transmission of sensor data according to the specification of remote sensor `rs` along the lines of Fig. 9.
- `UpdateRemoteSensor(rs)` processes the received data on remote sensor `rs` and updates the remote sensor view accordingly (see Fig. 10).

The semantical interpretations of the statements mentioned above are very similar to those presented in the Definitions 10–12 and are therefore not presented in detail. Besides that, these elements would very rarely be used explicitly within agent process definitions. Instead, they are used by implicit background processes as described in Section 7.3.

7.3. Transparent sensing infrastructure

With the process elements introduced in the end of the last section, it would be possible to treat local and remote sensing as explicit agent tasks in the same manner as communication with other agents. However this would lead to process definitions that mix core agent logic with infrastructure elements. In fact, it is closer to common realistic agent architectures to place sensing facilities into a separate layer and make the sensed information available in a transparent way. To achieve that, the SALMA framework automatically installs some *background processes*:

For each *local sensor*, an update process is installed at each agent that owns such a sensor. This process repeatedly executes `Sense`, i.e. initiates the sensing process. The actual *sensor view fluent* is updated after a delay that depends on the probability distributions that are set up for the specific sensor. By default, the update process is set up as *periodic* with a fixed period that is set up once in the simulation setup. However, other scheduling schemes are also possible, e.g. to reflect adaptive sensing strategies that try to optimize energy consumption.

Similarly, a *transmission process* is installed for each remote sensor at each agent of the *remote data source type*. This process transmits the most recent value of the configured local sensor to the remote sensor data sinks in the ensemble. By default, the transmission process is configured as periodic with a fixed period that would normally be set significantly longer than for the monitored local sensor. In addition to the transmission process, a reception process is created for each *data sink agent*, i.e. each owner of a remote sensor. Here, received remote sensor data is processed as described in Fig. 10. Other than the other background processes, the reception handler is installed as a triggered process that is executed as soon as new data becomes available.

With the implicit background processes in place, the modeler can access the most recent values of local and remote sensors in the same way as directly available fluents. For instance, Fig. 15 is taken from the e-mobility example and shows an excerpt from the PLCS agent process that handles reservation requests by vehicles. Here, the condition `freeSlotsL(self) > 0` is used to test whether free slots are available at a PLCS, and hence to decide whether to accept or reject a request.

8. Statistical model checking for information transfer

Once a system model has been created and configured in the way described above, SALMA's statistical model checker can be used to approximately assert system properties based on simulation results (cf. Section 4). These properties are defined using SALMA's property specification language whose core grammar is summarized below:

Definition 13 (*General SALMA-PSL Expression*)**.** Let c represent a constant, x a variable of any type, T any entity type with finite domain, $t \in \mathbb{N}_0$ a number of time units, $\sim$ a comparison operator ($<, \leq, =, \geq, >, \neq$), $\odot$ an arithmetic operator, α an action or event, p a general predicate, and f a general function. Furthermore, let F_F and F_R stand for a functional and relational fluent, respectively, and let F comprise both of these categories. Then, an expression Φ of the SALMA property specification language (SALMA-PSL) is defined recursively as follows:

$$\begin{aligned}\Phi :=\ & true \mid false \mid \tau_1 \sim \tau_2 \mid not(\Phi) \\ & \mid and(\Phi_1, \ldots, \Phi_n) \mid or(\Phi_1, \ldots, \Phi_n) \\ & \mid implies(\Phi_1, \Phi_2) \mid p(\tau_1, \ldots, \tau_n) \\ & \mid F_R(\tau_1, \ldots, \tau_n) \mid occur(\alpha(\tau_1, \ldots, \tau_n)) \\ & \mid hasChanged(F(\tau_1, \ldots, \tau_n)) \mid \\ & forall(x : T, \Phi) \mid exists(x : T, \Phi) \\ & \mid eventually(t, \Phi) \mid always(t, \Phi) \mid until(t, \Phi) \\ \tau :=\ & c \mid x \mid ?[f(\tau_1, \ldots, \tau_n) \mid F_F(\tau_1, \ldots, \tau_n) \mid \tau_1 \odot \tau_2 \mid (\tau) \\ & \mid changeTime(F(\tau_1, \ldots, \tau_n)) \mid \\ & lastOccurrence(\alpha(\tau_1, \ldots, \tau_n)).\end{aligned}$$ ■

```
...
If("freeSlotsL(self) > 0", [
    Act("add_reservation", [SELF, vehicle]),
    Send("reservation", Term("rresp", SELF, True), "plcs", vehicle, "veh")],
    [ # ELSE
    Send("reservation", Term("rresp", SELF, False), "plcs", vehicle, "veh")])
...
```

Fig. 15. Excerpt from reservation processing process of plcs agents.

The semantics of the logical connectives and of the common temporal operators *always*, *eventually* and *until* is equivalent to the common interpretation of time-bounded LTL formulas found, e.g., in [11]. The significant difference to the traditional use of LTL is that the SALMA-PSL supports expressions in a multi-sorted first-order predicate logic. Since the quantifiers are restricted to finite entity domains, they can be interpreted simply as conjunctions and disjunctions, respectively. Besides that, the SALMA-PSL adds the special predicates *occur* and *hasChanged* as well as the functions *lastOccurrence* and *changeTime*. The two predicates allow testing whether an action has occurred or a fluent has changed in the current time step. Similarly, the mentioned functions return the time point of the last occurrence of an action or event, or of the last change of a fluent.

Altogether, the SALMA-PSL makes it possible to refer directly to entities and agents, and to reason about their properties and relations. Since messages and connectors are also represented as entities, this means that all elements of communication and sensing processes can be examined with fine granularity. Additionally, the SALMA-PSL provides a set of specialized functions and predicates that can be used together with the fluents defined in Section 6.2 and Appendix to create an intuitive way for reasoning about the content of the information transfers, e.g.:

- *messageSent*$(chan, src, r_s, dst, r_d, msg)$ is a predicate that is true if, in the current time step, a message with content *msg* has been sent from source agent *src* to destination *dst* on channel *chan* with the given source and destination roles r_s an r_d.
- *messageAvailable*$(chan, dst, r_s)$ is a predicate that is true if the incoming message queue of agent *dst* for channel *chan* contains at least one message that addresses *dst* with role r_s.
- $src(m)$, $dest(m)$, and $con(m)$ are functions that return the source, destination, and the connector of a given message.
- *age* $(sen, a, [a_r])$ is a function that returns the age of the most recent value for the local or remote sensor *sen* of agent *a*. If *sen* is a remote sensor, then *age* refers to the value transmitted by the remote agent a_r.

Examples for the use of the SALMA property specification language in the context of information transfer processes can be found in Fig. 16. The invariant P1 requires that when any vehicle agent sends an assignment request to the SAM, it will not take longer than 100 time units until a target PLCS has been set. The question marks in the predicate `messageSent` serve as wild-card arguments for pattern matching, which achieve here that the recipient of the message, the involved roles, and the content of the message are ignored.

As other examples, P2 and P3 are invariants that define, for all measurements acquired by the remote sensor `freeSlotsR`, a maximum value age of 10 time units and a maximum deviation of 1 from the original sensor `freeSlotsL`.

Finally, property P4 demonstrates how the *content* of a message can be used directly in SALMA-PSL expressions. The property, which refers to the example in Fig. 15, states that every time a reservation is made by a PLCS agent (action `add_reservation`), a positive acknowledgment message must be sent within 10 time units. In order to test that, the content of the sent message has to be compared to the expected content according to Fig. 15 (`rresp(p, true)`).

Altogether, it can be seen that the ability to use typical features from first-order logic, like quantified variables and term unification, greatly facilitates the expression of complex properties. In particular this has a great benefit for properties that reason about the details of communication and sensor data propagation.

9. First experiments and preliminary evaluation

In order to test the presented approach and its integration in the SALMA toolkit, we implemented a reduced version of the scenario introduced in Section 2. It contains only a simple mock-up version of the optimization mechanism but realizes the full communication structure according to the approach presented in this paper. Both the SALMA toolkit and the model are available at the SALMA website.[2]

In the e-mobility model, the map, on which vehicles are moving around, is represented by a directed weighted graph with three different node types: crossings, point of interests (POIs), and parking lots with charging stations (PLCS). The weighted edges represent *roads* with a certain length, that lead from a start node to an end node. Fig. 17 shows the undirected graph that was modeled as a GraphML [16] file and used to derive the map which in turn was used for the concrete experiment described below. The undirected edges were translated to two directed edges in opposite direction with lengths that were derived from the geometrical information stored in the original file.

Vehicles are modeled as agents that move around the map to reach a certain point of interest (POI) that is randomly assigned to them at the beginning of the simulation. The vehicle agent then communicates with the super autonomous manager (SAM) and the PLCS agents according to Fig. 1. First it requests an assignment of a PLCS that is as close as possible to the vehicle's POI. In the current version, the SAM agent only uses a trivial optimization strategy that merely assigns the first PLCS with free slots. After the assignment has been made, the vehicle requests a reservation at the assigned PLCS. Once this reservation has been granted, the vehicle agent sets its target PLCS and calculates an optimal *route* to the assigned PLCS.

The communication between vehicles, the SAM, and PLCS agents, is performed by agent processes like the ones shown in Figs. 14 and 15. In contrast, the actual vehicle movement is not performed by explicit agent processes but encoded directly in the

[2] www.salmatoolkit.org.

```
P1 = forall(v:vehicle, implies(messageSent(v, assignment, ?, ?, ?, ?),
        eventually(100, currentTargetPLCS(v) \= none)))
P2 = forall(s:plcssam, forall(p:plcs, age(freeSlotsR, s, p) =< 10))
P3 = forall(s:plcssam, forall(p:plcs, abs(freeSlotsR(s, p) - freeSlotsL(p)) =< 1))
P4 = forall(p:plcs, forall(v:vehicle, implies(
        occur(add_reservation(p, v)),
        eventually(10, messageSent(p, reservation, plcs, v, vehicle,
                rresp(p, true) )))))
```

Fig. 16. Example SALMA-PSL properties of information transfer processes.

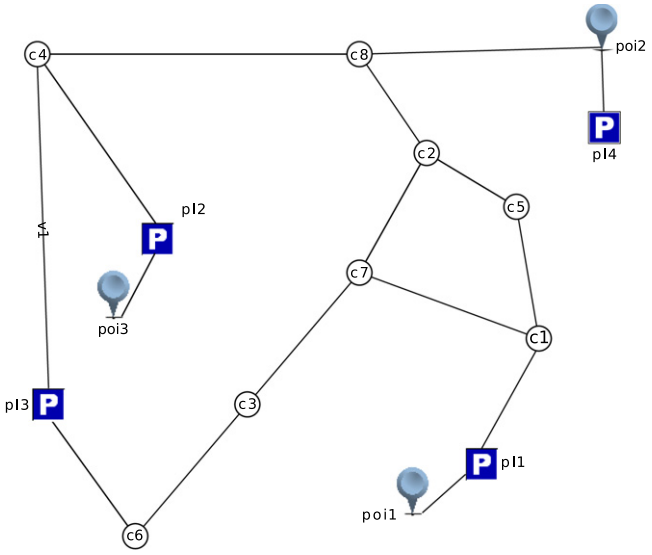

Fig. 17. Map used in the e-mobility experiment.

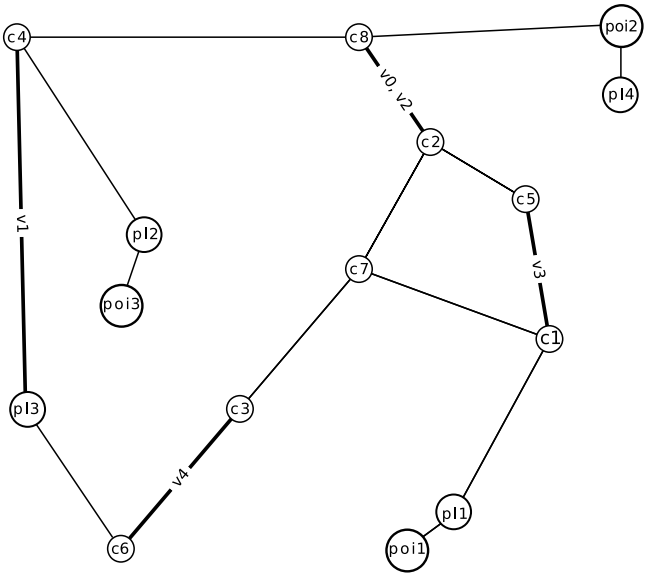

Fig. 18. Visualization of one step in e-mobility simulation.

situation calculus model of the simulation. There, the position of each vehicle is represented either by a current node or by the current road the vehicle is driving on. The current route followed by a vehicle is given as a list of roads that in sequence lead from the current position to the target PLCS. A more exact location on the road is actually not represented directly in the model. Instead, the road length is used as a factor for the delay with which events are scheduled that update make the vehicle move to the next route segment. This style of sparse simulation with pure event-scheduling is very well suited for the discussed example, in which the focus is set on the information transfer aspects rather than vehicle behavior or traffic.

The simulation of the e-mobility example can be run in three different modes: visualization, estimation, and hypothesis test. In visualization mode, the simulation is performed until all vehicles have arrived at their target PLCS. Meanwhile, the positions of all vehicles on the map are visualized in each step as annotations on the map. For instance, Fig. 18 shows a step of a simulation run with five vehicles (`v0`–`v4`) where all vehicles are currently located on roads, which is shown by labeling edges with the ids of the vehicles on the corresponding road. Additionally, textual information about the simulation state in each step is written to a logfile. Fig. 19 contains the output for the 102*nd* step of an e-mobility simulation run. Lines 2–4 show the actions and events that were performed in this step. It can be seen that *transferStarts*-messages occurred for the messages 231 and 228, both without an error term. The messages that currently exist in the system are listed from line 5 to line 18. It turns out that both messages belong to local sensors of the type *freeSlotsL*, which is used by PLCS agents to "measure" the. Each message line in the log output contains the message specification and the *transferred content*, separated by a colon. In this case, the transferred content is set now for both sensor messages. In fact, the last four lines of the output each show the confirm that the transferred values coincide with the true numbers of free slots of the PLCS *pl*1 and *pl*3, to which the messages belong. This conforms closely to the information transfer model for local sensing described in Fig. 8.

When the simulation is executed in estimation or hypothesis test mode, then it is run until all vehicles have been granted a reservation for their target PLCS while the invariant `P1` from Fig. 16 is satisfied. The probability that a simulation run succeeds for `P1` depends on a wide range of factors within the model. For example, in one experiment we used a fixed setup with the map from Fig. 17 and 5 vehicles. We only varied the time limit in the `until` operator of property `P1` from 5 time units to 145 time units in steps of 5. For each of the 28 configurations, 50 simulation runs were performed. Fig. 20 shows the proportion p of successful simulation runs for the different time limits. It can be seen that there is no chance for success below 55 time units while on the other hand success seems to be certain for time limits over 145. The buckle at around 100 time units shows that the success of a simulation run also depends on other random factors.

Another question that we examined was the runtime of the simulations and its dependence on the model complexity. In

```
Step 102 (t = 116)
   Actions: [('transferStarts', (231, None)), ('enterNextRoad', ('v4',)),
            ('transferStarts', (228, None)), ('arriveAtRoadEnd', ('v3',))]
      #227: msg('freeSlotsL', 'sensor', 'pl4', ()) : None
      #228: msg('freeSlotsL', 'sensor', 'pl3', ()) : 10
      #229: msg('freeSlotsL', 'sensor', 'pl2', ()) : None
      #231: msg('freeSlotsL', 'sensor', 'pl1', ()) : 10
   v4: r19(c3-c6=193) - ['r29(c6-pl3=170)'] - pl3 / None
   v0: c2 - ['r24(c2-c8=128)', 'r6(c8-poi2=331)', 'r32(poi2-pl4=60)'] - pl4 / None
   v2: r24(c2-c8=128) - ['r6(c8-poi2=331)', 'r32(poi2-pl4=60)'] - pl4 / None
   v3: c5 - ['r26(c5-c1=124)', 'r2(c1-pl1=193)'] - pl1 / None
   v1: pl3 - ['r10(pl3-c4=320)', 'r16(c4-c8=431)', 'r6(c8-poi2=331)',
                    'r32(poi2-pl4=60)'] - pl4 / None
   pl1:plcs: real = 10, local = 10, remote = 10
   pl2:plcs: real = 10, local = 10, remote = 10
   pl4:plcs: real = 10, local = 10, remote = 10
   pl3:plcs: real = 10, local = 10, remote = 10
```

Fig. 19. Log output for one step of an e-mobility simulation run.

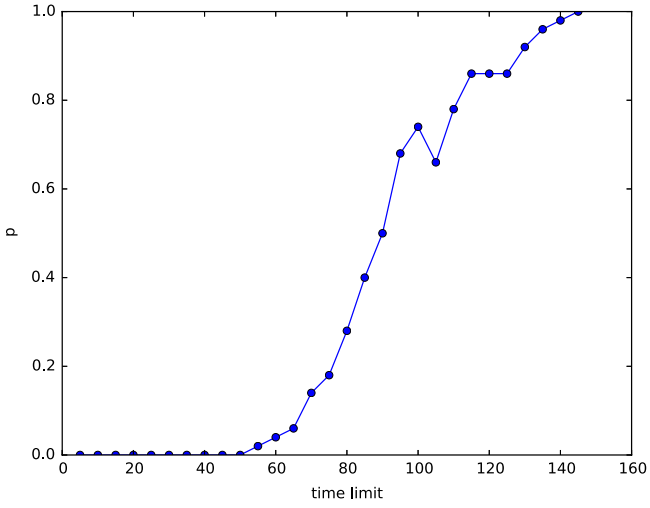

Fig. 20. Proportion of successful simulation runs for varying time limits.

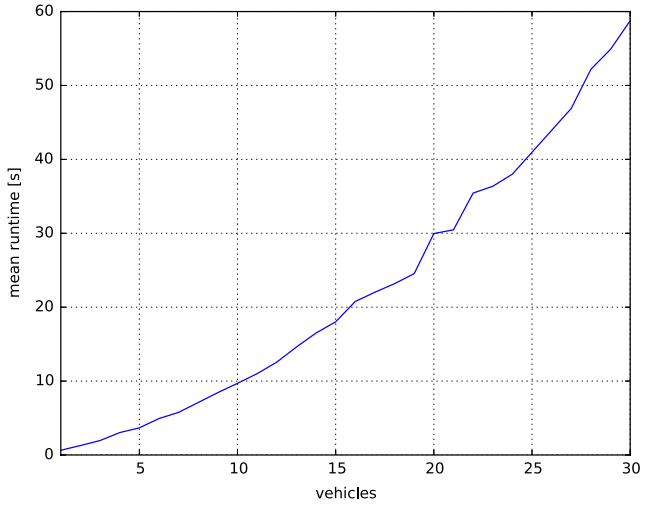

Fig. 21. Mean simulation runtimes for varying vehicle numbers.

particular, it is expected that an increasing number of agents will lead to an increased number of transferred messages, which increases the simulation complexity since messages are treated as entities. To analyze this effect, we set the time limit to 500 so that every simulation run is guaranteed to succeed and varied the number of vehicles from 1 to 30. Simulation batches of 20 repetitions for each vehicle number setting were performed on a system with Intel® Core™ i7-870 Processor with 2.93 GHz. The results are shown in Fig. 21. It reveals a dependence that is nearly linear with a peak of about 59*s* for 30 vehicles. At the same time, the memory consumption, which is not shown here, was not affected significantly. This implies that it is at least possible to handle simulations with very large agent numbers and parallelize the gathering of results by running simulations on multiple nodes. By this, enough data can be collected to achieve the desired level of exactness for statistical model checking.

Not only the number of vehicles can be varied in the simulation but several other parameters, including PLCS capacity, and the probability distributions for the information transfer events (cf. Section 6). Additionally, the Python function that realizes the PLCS assignment can be replaced. Thus, different optimization schemes can be tested and the impact of factors like delays or transmission errors can be analyzed. A detailed evaluation of the model is still ongoing and beyond the scope of this paper. However, experiences gained through experimentation and testing show that our information transfer model is well applicable also for complex communication scenarios. In particular, our proposed declarative high-level language has proven to be able to significantly improve clarity and conciseness of the model. For instance, the declarative part related to communication and sensing in the model mentioned above requires only about 30 lines in the style of the examples in Section 7.1. In contrast, the corresponding part of a functionally equivalent model that employs a direct axiomatization instead of the high-level abstractions, contains 15 fluents and 21 actions and events together with their associated axioms, which requires more than 200 lines of Prolog code.

10. Related work

As the primary contribution of the paper lies in providing high-level constructs for modeling and distributed communication of agents with realistic semantics, the related work can be split to three main areas: (1) communication models for situation calculi, (2) coordination languages for distributed agents, and (3) simulations of network communication in distributed systems.

From the perspective of the communication model for situation calculi, the information in the situation calculus has traditionally been viewed from an epistemic perspective, i.e. as knowledge that agents gain through (communication) actions. Surprisingly enough, there are no systematic approaches that introduce in the frame of a situation calculus a communication model reflecting properties of real-life computer networks. In this respect, a partial approach is provided by [17], where the epistemic model has been extended to model inter-agent communication by means of channels in a similar way as in our model described above. However, neither time nor stochastic effects are covered. In contrast to that, the approach presented in [18] combines the epistemic model with time and concurrency and allows reasoning about time-related aspects like the age of measurements.

Unlike the approaches mentioned above, our model does not consider knowledge in the epistemic sense but leaves the interpretation of transferred information to the agent processes. While we think that this perspective is better suited in the particular context of cyber–physical systems, it would be possible to combine both views in a straight-forward way.

Another perspective is provided by the coordination languages for distributed agents. These approaches are not based on the situation calculus. Rather they provide their own process algebras to define the behavior of distributed agents. These approaches are primarily represented by SCEL [19] and its indirect predecessor KLAIM [20]. They feature communication based on structure knowledge exchange via tuple-spaces. The process algebra they use for specification of the behavior of agents is typically based on or at least inspired by π-calculus.

In their original form the coordination languages do not consider time and communication uncertainty, which makes them unfit for realistic modeling of network communication. However extensions exist that feature these concerns forming a relatively large family of stochastic process algebras like TIPP [21], PEPA [22], EMPA [23,24], stochastic π-calculus [25,26], StoKlaim [27] or a unifying framework by Nicola et al. [28]. Targeting the framework of SCEL, which is closest to our approach, the ideas of stochastic process algebras are well integrated in [15], where the authors introduce a stochastically timed process calculus that is centered around predicate-based communication. Like our model, the most detailed semantical variant they describe distinguishes between a preparation and a transmission phase and allows assigning separate probability distributions for delays and errors to each of them. However, since the semantics is based on continuous time Markov chains (CMTC), only exponential distributions can be used and delays or errors are effectively determined at the start of each phase. This can be too coarse-grained in very dynamic situations, e.g. when the movement of agents has significant effect.

Targeting specifically the analysis of communication and processing in distributed systems, network simulators, e.g. OMNet++ [29], ns-3 [30], provide very accurate estimates. The simulators feature an agent-like approach, where the simulated network consists of a number of modules (representing end-devices and network components) mutually communicating by exchanging messages. These modules are triggered by a discrete simulator based on timing needed for message processing, communication latencies, etc. To achieve simulation of environment when agent mobility is involved, the network simulators can be integrated with mobility and traffic simulators, e.g. MATSim [31], Sumo [32].

Compared to our approach, network simulators provide significantly more precise estimates in terms of network communication latencies. However, this comes for a price of significantly more complex (and consequently by few orders of magnitude slower) simulation. Further, compared to the situation calculus and the high-level interaction patterns, the simulators lack in providing abstract enough architectural and coordination perspective. Further, there is no option of LTL-based or similar verification of traces generated by OMNet++ or ns-3.

The problem of too low level of abstraction and extensive simulation time of the network simulators is typically addressed by high-level communication and processing models, such as (layered) queueing networks [33]. These however do not provide means for specification of agent-behavior.

11. Conclusion

We have presented a new logic-based approach for modeling channel-based communication, sensing, and other kinds of information transfer within cyber–physical multi-agent systems. The proposed high-level language provides means to embrace the stochastic nature of these systems, like transmission delays and errors. At the same time it has a precise formal semantics based on the first-order logic situation calculus. Therefore, it can be integrated in existing logic-based approaches for verification and validation, in particular SALMA, a framework for simulation and statistical model checking we have introduced earlier in [1]. A major advantage of this combination is that SALMA's property specification language, based on a first-order temporal logic, allows fine-grained reasoning about the inner details of information transfer processes.

Experiences gained in experiments with a mid-sized case study show that our approach offers great flexibility with respect to the level of detail and accuracy with which both the system model and corresponding requirements are formulated. Altogether, SALMA exhibits great potential to contribute to practicable verification and validation of self-adaptive cyber–physical multi-agent systems.

Acknowledgment

This work has been partially sponsored by the EU project ASCENS, FP7 257414.

Appendix. Axiomatization of the information transfer model

Definition 14 (*Message Life Cycle*)**.** The state of a message is defined by two separate fluents that define whether it is waiting to be transferred or actually transferring.

$$\begin{aligned} &awaitingTransfer(m, do(a, s)) \equiv a = requestTransfer(m) \\ &\quad \vee ((\nexists \varepsilon.\ a = transferStarts(m, \varepsilon)) \wedge a \neq transferFails(m) \\ &\quad \wedge awaitingTransfer(m, s)) \end{aligned} \tag{A.1}$$

$$\begin{aligned} &transferring(m, do(a, s)) \equiv (\exists \varepsilon.\ a = transferStarts(m, \varepsilon)) \\ &\quad \vee ((\nexists \varepsilon.\ a = transferEnds(m, \varepsilon)) \wedge a \neq transferFails(m) \\ &\quad \wedge transferring(m, s)). \end{aligned} \tag{A.2}$$

The action *requestTransfer* is the entry point with which agents initiate the information transfer.

$$\begin{aligned} &Poss(requestTransfer(m), s) \equiv \neg awaitingTransfer(m, s) \\ &\quad \wedge \neg transferring(m, s). \end{aligned} \tag{A.3}$$

The actual start of the information transfer is marked by the event *transferStarts*.

$$Poss(transferStarts(m, \varepsilon), s) \equiv awaitingTransfer(m, s). \tag{A.4}$$

A message has been transferred successfully when *transferEnds* occurs. However, source messages of multicast or remote sensor transmissions are not ended explicitly this way but are removed when all of their multicast copies have arrived or failed (cf. Definition 21).

$$\begin{aligned} &Poss(transferEnds(m, \varepsilon), s) \equiv transferring(m, s) \\ &\quad \wedge type(m) \notin \{multicastSrc, remoteSensorSrc\}. \end{aligned} \tag{A.5}$$

An information transfer can fail at any time after the transfer has been requested. For multicast and remote sensor transfers, the same argument holds as for *transferEnds*.

$$\begin{aligned} &Poss(transferFails(m), s) \equiv (awaitingTransfer(m, s) \\ &\quad \vee transferring(m, s)) \\ &\quad \wedge type(m) \notin \{multicastSrc, remoteSensorSrc\}. \end{aligned} \tag{A.6}$$

Definition 15 (*Channel-Based Message Transmission Content*)**.** The content of active messages that are *channel-based* (including remote sensors) is defined by the fluent C_{trans} whose content is set when a *transferStarts* event for the message occurs. For remote sensors, the content is received from the corresponding source sensor, while for directly sent messages, the content is received from the message constant C_{out} that is set during a Send action.

$$\begin{aligned} &\forall m, s.\ domain(Msg, s) \Longrightarrow C_{trans}(m, do(a, s)) = y \\ &\quad \equiv \big(\exists \varepsilon. a = transferStarts(m, \varepsilon) \\ &\qquad \wedge ((m \in Msg_{remSenSrc} \\ &\qquad \wedge locSen(connector(m)) = f_l \\ &\qquad \wedge y = f_l(src(m), s) + \varepsilon) \\ &\qquad \vee (m \notin Msg_{remSenSrc} \\ &\qquad \wedge y = C_{out}(m, s) + \varepsilon))\big) \\ &\qquad \vee \big(\nexists \varepsilon. a = transferStarts(m, \varepsilon) \\ &\qquad \wedge y = C_{trans}(m, s)\big). \end{aligned} \tag{A.7}$$

For multicast destination copies, it holds that their content is always equal to the content of the original message. Deviations that are inflicted on individual message paths are aggregated when the message arrives at the destination (cf. Definition 16).

$$\begin{aligned} &\forall m \in Msg_{multicastDest}. \forall s.\ m \in domain(Msg, s) \\ &\quad \Longrightarrow C_{trans}(m, s) = C_{trans}(srcMsg(m), s) \end{aligned} \tag{A.8}$$

Definition 16 (*Incoming Message Queue*)**.**

$$\begin{aligned} &queue_{in}(c, do(a, s)) = y \\ &\quad \equiv (\exists m, \varepsilon.\ a = transferEnds(m, \varepsilon) \\ &\qquad \wedge chan(m) = c \wedge y = \{msg(sender(m), srcRole(m), \\ &\qquad dest(m), destRole(m), \\ &\qquad time(m, s), C_{trans}(m, s) + \varepsilon)\} \cup queue_{in}(c, s) \\ &\qquad \vee ((\exists ag, r.\ a = cleanQueue(ag, c, r)) \\ &\qquad \wedge y = \big\{e \mid e \in queue_{in}(c, s) \wedge dest(e) \neq ag \\ &\qquad \wedge destRole(e) \neq r\big\}) \\ &\qquad \vee ((\forall m, \varepsilon.\ a \neq transferEnds(m, \varepsilon)) \\ &\qquad \wedge (\forall ag, r.\ a \neq cleanQueue(ag, c, r)) \\ &\qquad \wedge y = queue_{in}(c, s)). \end{aligned} \tag{A.9}$$

Definition 17 (*Local Sensor View*)**.** Let *sen* be the name of a local (direct) sensor. Then the model contains a corresponding fluent with the same name that stores the result of the last measurement as defined below. The notation «sen» is used below as a placeholder for the actual sensor name.

$$\text{«sen»} : Agent \times Sit \rightarrow T \quad \in \quad Fluents \tag{A.10}$$

$$\begin{aligned} &\text{«sen»}(ag, do(a, s)) = y \\ &\quad \equiv (\exists m, \varepsilon.\ \Phi(ag, a, m, \varepsilon) \wedge y = Sen_{trans}(m, s) + \varepsilon) \\ &\quad \vee ((\nexists m, \varepsilon. \Phi(ag, a, m, \varepsilon)) \wedge y = \text{«sen»}(ag, s)) \end{aligned} \tag{A.11}$$

where

$$\begin{aligned} &\Phi(ag, a, m, \varepsilon) \equiv a = transferEnds(m, \varepsilon) \wedge con(m) = \text{«sen»} \\ &\quad \wedge src(m) = ag. \end{aligned} \tag{A.12}$$

Definition 18 (*Transmitted Local Sensor Data*)**.** The information that is transferred by a local sensor is defined in the fluent sen_{trans} and derived from the sensor's source fluent.

$$\begin{aligned} &\forall m, s.\ m \in domain(Msg, s) \Longrightarrow sen_{trans}(m, do(a, s)) = y \\ &\quad \equiv (\exists \varepsilon. \Phi(m, a, \varepsilon) \wedge f = srcFluent(sensor(m)) \\ &\qquad \wedge y = f(src(m), s) + \varepsilon) \\ &\qquad \vee ((\nexists \varepsilon. \Phi(m, a, \varepsilon)) \ \wedge \ y = sen_{trans}(m, s)) \end{aligned} \tag{A.13}$$

where

$$\Phi(m, a, \varepsilon) \equiv a = transferStarts(m, \varepsilon) \wedge m \in Msg_{locSen}. \tag{A.14}$$

Definition 19 (*Remote Sensor View*)**.**

$$\begin{aligned} &\text{«rsen»}(a_d, a_s, do(a, s)) = y \\ &\quad \equiv (a = updateRemoteSensor(a_d, \text{«rsen»}) \\ &\qquad \wedge \exists m.\ m \in queue_{in}(\text{«rsen»}, s) \\ &\qquad \wedge dest(m) = a_d \wedge src(m) = a_s \\ &\qquad \wedge (\nexists m'.\ m' \in queue_{in}(\text{«rsen»}, s) \\ &\qquad \wedge tstamp(m') > tstamp(m)) \\ &\qquad \wedge y = content(m)) \\ &\qquad \vee (a \neq updateRemoteSensor(a_d, \text{«rsen»}) \\ &\qquad \wedge y = \text{«rsen»}(a_d, a_s, s)). \end{aligned} \tag{A.15}$$

Definition 20 (*Multicast Copy*)**.** The predicate *multicastCopy* relates a source message sent on a multicast or remote sensor channel to a copy of this message for a given destination.

$$multicastCopy \subset Msg \times Msg \times Agent \tag{A.16}$$

$$\begin{aligned} &multicastCopy(m', m, d) \equiv con(m) = con(m') \\ &\quad \wedge m' \in Msg_{multicastDest} \wedge dest(m') = d \wedge src(m) = src(m') \\ &\quad \wedge srcMsg(m') = m \wedge destRole(m') \\ &\quad = oppositeRole(con(m), srcRole(m)). \end{aligned} \tag{A.17}$$

Definition 21 (*Active Message Domain*)**.** The representation of messages as entities requires that they can be created and removed dynamically as effect to actions and events. Unlike traditional realizations of the situation calculus, SALMA supports this by using a special (meta-)fluent *domain*(*sort*) to store the sets of entities that manifest the current *domains* of all sorts in the model. Creation and destruction of entities can therefore be controlled through regular successor state axioms. For the sort *Message*, the following

SSA is defined:

$$
\begin{aligned}
&domain(Msg, do(a, s)) = D \\
&\quad \equiv (\exists m, \varepsilon.\ \Phi_1(a, m, \varepsilon) \\
&\qquad \wedge ((m \in Msg_{multicastDest} \wedge D \\
&\qquad = domain(Msg, s) \backslash \{m, srcMsg(m)\}) \\
&\qquad \vee (m \notin Msg_{multicastDest} \wedge D = domain(Msg, s) \backslash \{m\}))) \\
&\qquad \vee (\exists m, \varepsilon.\ \Phi_2(a, m, \varepsilon) \\
&\qquad \wedge D = domain(Msg, s) \\
&\qquad \cup \{m' \mid d \in Agents \wedge multicastCopy(m', m, d) \\
&\qquad \wedge ensemble(con(m), src(m), d)\}) \\
&\qquad \vee ((\nexists m, \varepsilon.\ \Phi_1(a, m, \varepsilon) \vee \Phi_2(a, m, \varepsilon)) \wedge D \\
&\quad = domain(Msg, s))
\end{aligned} \tag{A.18}
$$

with

$$
\begin{aligned}
\Phi_1(a, m, \varepsilon) &\equiv a = transferEnds(m, \varepsilon) \vee a \\
&\quad = transferFails(m)
\end{aligned} \tag{A.19}
$$

$$
\begin{aligned}
\Phi_2(a, m, \varepsilon) &\equiv a = transferStarts(m, \varepsilon) \\
&\quad \wedge m \in Msg_{multicastSrc} \cup Msg_{remoteSensorSrc}.
\end{aligned} \tag{A.20}
$$

References

[1] C. Kroiß, Simulation and statistical model checking of logic-based multi-agent system models, in: 8th International Conference on Agent and Multi-Agent Systems: Technologies and Applications, KES-AMSTA 2014, 2014, pp. 151–160.
[2] A. Legay, B. Delahaye, S. Bensalem, Statistical model checking: An overview, in: Runtime Verification, Springer, 2010, pp. 122–135.
[3] R. Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT press, 2001.
[4] E.A. Lee, Cyber physical systems: Design challenges, in: 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing, ISORC 2008, 2008, pp. 363–369.
[5] R.B. Scherl, H.J. Levesque, Knowledge, action, and the frame problem, Artif. Intell. 144 (1) (2003) 1–39.
[6] T. Bureš, et al., A life cycle for the development of autonomic systems: the e-mobility showcase, in: 3rd Workshop on Challenges for Achieving Self-Awareness in Automatic Systems, IEEE, 2013, pp. 71–76.
[7] H.J. Levesque, et al., Golog: A logic programming language for dynamic domains, J. Log. Program. 31 (1) (1997) 59–83.
[8] M. Fowler, Domain-Specific Languages, Pearson Education, 2010.
[9] M. Wooldridge, An Introduction to Multiagent Systems, John Wiley & Sons, 2009.
[10] J. Banks, J.S. Carson II, B.L. Nelson, D.M. Nicol, Discrete-Event System Simulation, fourth ed., Prentice Hall, Upper Saddle River, NJ, 2004.
[11] A. Pnueli, The temporal logic of programs, in: Foundations of Computer Science, SFCS'77, IEEE Computer Society, Washington, DC, USA, 1977, pp. 46–57. http://dx.doi.org/10.1109/SFCS.1977.32.
URL http://dx.doi.org/10.1109/SFCS.1977.32.
[12] A. Wald, et al., Sequential tests of statistical hypotheses, Ann. Math. Stat. 16 (2) (1945) 117–186.
[13] L.D. Brown, T.T. Cai, A. DasGupta, Interval estimation for a binomial proportion, Stat. Sci (2001) 101–117.
[14] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, Cambridge, Massachusetts, 2008.
[15] D. Latella, et al. Stochastically timed predicate-based communication primitives for autonomic computing, Tech. rep., QUANTICOL Project (2014).
[16] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, M.S. Marshall, Graphml progress report structural layer proposal, in: Graph Drawing, Springer, 2002, pp. 501–512.
[17] D. Marcu, et al. Distributed software agents and communication in the situation calculus, in: International Workshop on Intelligent Computer Communication, 1995, pp. 69–78.
[18] R.B. Scherl, Reasoning about the interaction of knowledge, time and concurrent actions in the situation calculus, in: 18th International Joint Conference on Artificial Intelligence, IJCAI-03, 2003, pp. 1091–1098.
[19] G. Cabri, N. Capodieci, L. Cesari, R.D. Nicola, R. Pugliese, F. Tiezzi, F. Zambonelli, Self-expression and dynamic attribute-based ensembles in SCEL, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, in: Lecture Notes in Computer Science, vol. 8802, Springer, Berlin Heidelberg, 2014, pp. 147–163. URL http://link.springer.com/chapter/10.1007/978-3-662-45234-9_11.
[20] L. Bettini, V. Bono, R.D. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, B. Venneri, The klaim project: Theory and practice, in: C. Priami (Ed.), Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 2874, Springer, Berlin Heidelberg, 2003, pp. 88–150.
URL http://link.springer.com/chapter/10.1007/978-3-540-40042-4_4.
[21] N. Götz, U. Herzog, M. Rettelbach, Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras, in: Performance Evaluation of Computer and Communication Systems, Joint Tutorial Papers of Performance '93 and Sigmetrics '93, Springer-Verlag, London, UK, UK, 1993, pp. 121–146. URL http://dl.acm.org/citation.cfm?id=647339.721059.
[22] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, New York, NY, USA, 1996.
[23] M. Bernardo, R. Gorrieri, A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time, Tech. Report, University of Bologna, 1996.
[24] A. Aldini, F. Corradini, M. Bernardo, Stochastically timed process algebra, in: A Process Algebraic Approach to Software Architecture Design, Springer, London, 2010, pp. 75–124. URL http://link.springer.com/chapter/10.1007/978-1-84800-223-4_3.
[25] C. Priami, Stochastic π-calculus, Comput. J. 38 (7) (1995) 578–589. http://dx.doi.org/10.1093/comjnl/38.7.578.
URL http://comjnl.oxfordjournals.org/content/38/7/578.
[26] L. Cardelli, R. Mardare, Stochastic pi-calculus revisited, in: Z. Liu, J. Woodcock, H. Zhu (Eds.), Theoretical Aspects of Computing—ICTAC 2013, in: Lecture Notes in Computer Science, vol. 8049, Springer, Berlin Heidelberg, 2013, pp. 1–21. URL http://link.springer.com/chapter/10.1007/978-3-642-39718-9_1.
[27] R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, M. Massink, Model checking mobile stochastic logic, Theoret. Comput. Sci. 382 (1) (2007) 42–70. http://dx.doi.org/10.1016/j.tcs.2007.05.008.
URL http://dx.doi.org/10.1016/j.tcs.2007.05.008.
[28] R.D. Nicola, D. Latella, M. Loreti, M. Massink, A uniform definition of stochastic process calculi, ACM Comput. Surv. 46 (1) (2013) 5:1–5:35. http://dx.doi.org/10.1145/2522968.2522973.
URL http://doi.acm.org/10.1145/2522968.2522973.
[29] OMNeT++ discrete event simulator—home. URL http://omnetpp.org/.
[30] ns-3. URL http://www.nsnam.org/.
[31] Agent-based transport simulations | MATSim. URL http://www.matsim.org/.
[32] DLR – institute of transportation systems – SUMO—simulation of urban MObility.
URL http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/.
[33] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi, Enhanced modeling and solution of layered queueing networks, IEEE Trans. Softw. Eng. 35 (2) (2009) 148–161. http://dx.doi.org/10.1109/TSE.2008.74.

Christian Kroiß is a Ph.D. student and research assistant at the Department for Informatics of LMU Munich. During the recent years, he has been working on the EU project ASCENS (Autonomic Service-Component Ensembles).

Tomas Bures is Associate Professor at the Department of Distributed and Dependable Systems of Charles University in Prague. During the recent years, he has been working on the EU project ASCENS (Autonomic Service-Component Ensembles).