Towards a Formal Framework for Mobile, Service-Oriented Sensor-Actuator Networks

Helena Gruhn
Technische Universität Berlin
Berlin, Germany
helena.gruhn@tu-berlin.de

Sabine Glesner
Technische Universität Berlin
Berlin, Germany
sabine.glesner@tu-berlin.de

Service-oriented sensor-actuator networks (SOSANETs) are deployed in health-critical applications like patient monitoring and have to fulfill strong safety requirements. However, a framework for the rigorous formal modeling and analysis of SOSANETs does not exist. In particular, there is currently no support for the verification of correct network behavior after node failure or loss/addition of communication links. To overcome this problem, we propose a formal framework for SOSANETs. The main idea is to base our framework on the π-calculus, a formally defined, compositional and well-established formalism. We choose KLAIM, an existing formal language based on the π-calculus as the foundation for our framework. With that, we are able to formally model SOSANETs with possible topology changes and network failures. This provides the basis for our future work on prediction, analysis and verification of the network behavior of these systems. Furthermore, we illustrate the real-life applicability of this approach by modeling and extending a use case scenario from the medical domain.

1 Introduction

Mobility and independence are some of the major factors defining the quality of life. With rising age people often develop physical and mental diseases of different degree influencing their options for an independent and mobile life style. Service-oriented sensor-actuator networks (SOSANETs) [25] employed as assistant systems at home and en route can support the user by securing their everyday life. These networks are highly dynamic. New communication channels are built ad hoc, connecting the system with other previously unknown SOSANETs. These systems are health-critical and have to fulfill strong safety requirements. Up to now there exists no formal design framework for the verification and analysis of SOSANETs. The common approach is to evaluate network behavior by running test scenarios on prototypes or simulations. These tests do not cover the whole state space of the network and cannot ensure correct system behavior.
In this paper, we address the problem of modeling dynamic SOSANETs. We require a formal modeling framework which offers techniques to model topology changes like the establishing of new connections or the sudden disappearance of a component. Especially the interaction with components, which were unknown at design level, has to be representable. Furthermore, we require the framework to offer features for modeling basic functionalities of service-oriented architectures like service publishing. Additionally, the framework needs to be platform-independent to allow the representation of all kinds of nodes.
Our idea is to base our framework on the π-calculus [20]. The π-calculus is a well-established formalism for the modeling of compositional and concurrent systems. It provides primitives for the description and analysis of distributed infrastructures. Furthermore, it offers the modeling

B. Buhnova, L. Happe, J. Kofroň:
Formal Engineering Approaches to Software Components
and Architectures 2013 (FESCA'13)
EPTCS 108, 2013, pp. 49–62, doi:10.4204/EPTCS.108.4

foundation for the verification of correct network behavior after topology changes.
We choose KLAIM [3], a formal language based on the π-calculus, as a foundation for our approach. It enriches the formalism by providing concrete actions for the manipulation of data within the network. In this paper we evaluate the applicability of KLAIM, which was developed for Wide Area Networks (WAN), for the modeling of SOSANETs. We consider the usability of the privileged node coordinator processes provided within KLAIM for the modeling of anticipated and unexpected topology changes in service-oriented sensor-actuator networks. Additionally, we assess the language to judge its ability to represent basic functions of a service-oriented network. These are the service publishing, the service discover and the service request. Furthermore, we validate our observations by modeling an use case from the medical domain.
The mentioned case study is a driver assistant system for the support of elderly driver. We extend the existing closed system to an open network to match our requirements. The open net is capable of interacting dynamically with expected new network components, e.g. portable medical devices like a pulse meter, and completely unknown networks, e.g. driver assistant systems installed in cars of different brands.

The remainer of this paper is structured as follows. In Section 2 we discuss related work in the area of formal modeling and verification of sensor-actuator networks and other distributed systems. This is followed by an introduction to SOSANETs, the π-calculus and KLAIM, the basis of our design framework, in Section 3. Section 4 illustrates our use case scenario and its proposed extension to an open network. We present a model of the driver assistant system and show the possibilities and limitations of KLAIM as a modeling language for SOSANETs in its current version. Afterwards, Section 5 concludes this paper and gives an outlook on future work.

2 Related work

In this section we provide a discussion on literature related to our work. There is no formal modeling framework for SOSANETs to the best of our knowledge. Still, work on formal modeling and verification on Wireless Sensor Networks (WSN), Sensor-Actuator Networks (SANET), Wide Area Networks and distributed systems exist, which is related to our application field.

Today, it is common practice to check the behavior of WSNs and SANETs by running tests on a simulation or prototype of the network (e.g. [16],[25]). However, the verification of these networks is an emerging field of research. Some papers were published in recent years presenting approaches for the formal modeling and verification of WSNs, SANETs and other distributed systems.
Martinez and al. [17] present an approach which uses Colored Petri Nets (CPN) for the modeling of SANETs. Their design method allows the integration of different node architecture levels (CPU, operating system, middleware...). They make use of well-known Petri nets properties to verify the network behavior. The biggest drawback of their approach for our task is that they do not consider topology changes at all. Even the break-down of a wireless link, a common event in wireless SANETs, is not representable.
UM-RTCOM [10] is another modeling framework for SANETs. It is written in CORBA and is real-time component based. The communication via a channel is modeled as a tuple. The model can be used to analyze e.g. deadlock freedom or verify liveness properties. UM-RTCOM uses

components to capture the system behavior. The model declares for each component interfaces, the services which are offered by each component, and the connectivity of each component. The components can invoke each other based on event triggers, time triggers, or service requests [15]. UM-RTCOM biggest shortcoming for our goal is the absence of tools for handling topology changes.

Dearle et al. present the modeling language Insense[9],[2]. It uses a component-based model for WSNs. Components are concurrent and use synchronous communication. They can represent software or hardware entities of a WSN. They capsulate a specific behavior and have interfaces to interact with other components of the system. There are no dependencies between components. Insense supports topology changes. Channels can be connected or disconnected. New component instances may be dynamically created and executing instances may be stopped. This allows the rewiring and replacing of arbitrary components at runtime. In a first approach to show the correctness of Insense, Sharma et al. used Promela constructs for modeling and Spin for verification [26]. A big misfit with our requirements is that Insense is built on the Contiki operating system [11] making the whole modeling procedure platform dependent.

The Promela Model, described by Oleshchuk [22], was created to verify the following correctness properties in Spin: sensors are always connected and at least one communication way for each sensor exist. It is assumed that nodes are in a dynamic movement state resulting in unreliable communication channels. The routing protocol ensures that each node is aware of neighboring nodes. The physical location of a node is regularly updated to a central unit, called Location Manager. There is no central unit in our approach due to strong resource restrictions on mobile devices which are the core units of our mobile patient monitoring system.

Francalanza and Hennessy present in [12] an approach to model node and link failure in distributed programs. They use the distributed π-calculus framework and extend it by a ping construct for detecting and reacting on those failures. Their approach does not cover the dynamical addition of nodes during run-time and is only applicable to SOSANETs via abstract constructs describing detailed process behavior. KLAIM, a formal language first published by de Nicola et al. [6], was developed for modeling Wide Area Networks. A detailed description of KLAIM can be found in Section 3. Although it is possible to model topology changes with this language, the addition of unknown (at deployment time) components can not be represented due to the login/accept constructs used for adding new network nodes.

None of the described approaches fulfills all our requirements for a formal design framework for mobile service-oriented sensor-actuator networks.

3 Background

Our approach is based on work introduced in this section. First, we present SOSANETs explaining the differences to and advantages over general wireless sensor-actuator networks. Afterwards we give an introduction to KLAIM.

3.1 Service-oriented sensor-actuator networks

Sensor-actuator networks (SANETs) consist of nodes with sensing and/or actuation capabilities. Sensors observe their surroundings and collect data. Actuators make decisions and affect the environment by performing various actions [1]. The communication between nodes can be physical or wireless. Nodes may be stationary or mobile.

There are two widely-used SANET architectures. Application-specific SANETs are often developed to suit a highly-specialized utilization. These are greatly optimized networks with restricted reusability value due to limited resources. A more open approach is used for generic SANETs, which are not intended for a specific employment but require a generic code installation on every node. This results in unusable code on some nodes, e.g. actuator coordination on a node without an actuator, in spite of limited storage capabilities. These design limitations are reduced by using a service-oriented approach for SANET development.
Originally the term *service-oriented architecture* (SOA) refers to a logical set that consists of several large software component that together perform a task or a *service*. The paradigm is popular e.g. in the web software developers world (Web Services) [23]. However, it can be easily adapted for SANETs by including not only large complex services, but also simple ones. These could be data storage, routing or sensor reading.
Service-oriented sensor-actuator networks (SOSANETs) [25] realize this concept: every node publishes its own capabilities as a service. A service is a computational component with a unique network-wide identifier. It can be invoked asynchronously and more than one instance can be found within the whole net. Clients can invoke a service through queries which are either directed to a base station or to a specific node. There are two query types: task queries (e.g. reading of a sensor and returning the results) or event queries (e.g. request of a notifier message if an event occured).
The main benefit of this architecture is that the services can be loosely coupled. Only the interface has to be known to provoke a service which allows interaction of devices of diverse producers without revealing construction details. Complex services can be composed using independent SOSANETs through dynamic search and connectivity. This leads to high reusability rates of code and hardware reducing the network development cost. Furthermore, SOSANETs can be extended and maintained while on-line.

3.2 The π-calculus

The π-calculus is a simple but powerful model of computation for concurrent systems. It was first published 1992 in "A calculus of mobile processes" [21] by Robin Milner, Joachim Parrow and David Walker. The word mobility stands here for the movement of links in the virtual space of linked processes [20].
The π-calculus has two basic entities: names and processes. Processes are an abstraction of an independent thread of control. Names are the channels (alternative: communication links) processes use to communicate by sending and receiving messages [20]. The ability to pass channels as data along other channels which allows to model process mobility distinguishes the π-calculus from preceding process algebras.
The used syntax for processes, actions and names is presented in Table 1.

$$\begin{array}{llllll} \pi :: & = \bar{x}\mathrm{y} & \mid \mathrm{x(y)} & \mid \tau & & \\ \mathrm{P} :: & = \mathbf{0} & \mid \pi.\mathrm{P} & \mid P_1 | P_2 & \mid P_1 + P_2 & \mid !\mathrm{P} \mid (\nu \mathrm{z})\mathrm{P} \end{array}$$

Table 1: π-calculus syntax

There are three prefixes expressing the capabilities of actions. The first two represent the sending or receiving of a name. The third stands for a silent transition. The term π.P denotes that P cannot proceed until action π occurred. Respectively, $\bar{x}\mathbf{y.P}$ is an *output prefix* saying that P can send the name y via channel x and than proceed as P. The *input prefix* $\mathbf{x(y).P}$ states that P can receive any name via channel x before it behaves like P. The *unobservable prefix* $\boldsymbol{\tau}\mathbf{.P}$ can evolve invisible to P [20]. $P_1|P_2$ is a *parallel composition* of two processes, while $P_1 + P_2$ models the *choice* between two variants of behavior. **!P** is called *replication* denoting an infinite number of instances of P running in parallel and **0** is an *inert process* that can do nothing. The *restriction* $(\boldsymbol{\nu}\mathbf{z})\mathbf{P}$ ensures that z is a new channel restricted to P allowing a private interaction between components of P.

This short list of constructs is enough to represent all imaginable variations of concurrent behavior.

3.3 KLAIM

KLAIM (kernel-language for agent interaction and mobile computing) is an experimental language designed to model Wide Area Networks (WANs) composed out of diverse, mobile components. It was first published 1998 by de Nicola, Ferrari and Pugliese [6]. The language is based on the π-calculus and the coordination language *Linda* [5]. It extends the expressive π-calculus with predefined actions for data manipulation as found in Linda. KLAIM allows dynamic channel creation. The communication is asynchronous. Although links can be built on the fly communication partners have to be known in advance.

P : :	=	PROCESSES	N::	=	NETS
	nil	null process		**0**	empty net
	| act.P	action prefixing		$s ::_{\rho}^{S} P$	single node
	| $P_1 \mid P_2$	parallel composition		$N_1 \mid\mid N_2$	net composition
	| $P_1 + P_2$	choice			
	| X	process variable			

Table 2: KLAIM syntax for processes and nets

In Table 2 the KLAIM syntax for processes is listed. It follows the notations which are commonly used in process algebras and are based on Milner's CCS [19]. **nil** stands for a process that can not perform any actions. *act.P* states that after the execution of action act the process behaves like P. There are two ways to compose two processes P_1 and P_2: $P_1 \mid P_2$ stands for a parallel composition, $P_1 + P_2$ for a nondeterministic one. Furthermore, a syntax for nets is described. A net can be an empty net **0**. It can be a single node or the parallel composition of two nets N_1 and N_2 with a disjoint set of node sites.
Following syntactic categories are used from here on:

- *Sites*: identifier through which a node can be uniquly identified through the network (comparable to IP adresses).
- *Localities*: symbolic name for a site allowing to structure programs over distributed environments without knowing their precise allocation. The locality **self** is used by processes to refer to their execution site.
- ρ: is the allocation environment, a finite partial function mapping localities with sites.

- *Network node*: 4-tuple $s ::_{\rho}^{S} P$, where s is the site of the node, S the set of sites logged into s, P a set of concurrent processes running on s and ρ the allocation environment of s.

The comunication between processes is supported through multiple distributed **tuple spaces (TS)**. There is a tuple space on each node. The TS can be seen as a local repository for data and interactions. **Tuples (t)** are sequences of information items and are anonymous. They are retrieved from tuple spacing using pattern-matching mechanisms and are manipulated with four predefined actions *act*. These actions are:

- **out(t)**: produces a tuple by writing it into the TS
- **eval(t)**: creates new instances of processes for tuple evaluation and writes the results into the TS
- **in(t)**: reads and consumes a tuple from the TS
- **read(t)**: reads tuples non-destructively
- **bind(ℓ,s)**: enhances local allocation environments with new aliases for sites

There is a clear separation between the computational level and the net coordinator/ administrator level in KLAIM. OpenKLAIM [3], a dialect of KLAIM, introduces a new category of processes called *NodeCoordinators*. They can be interpretated as kind of superuser managing topology changes within the network. NodeCoordinators can perform privileged actions to create new nodes and establish and/or remove communication channels. These priveledged actions are listed below.

- **newloc**(s,$\mathbb{C}$) creates a new node in the network with the site s and installes a NodeCoordinator process $\mathbb{C}$ on the new node.
- **login**(ℓ) builts a connection between the performing node and the node with locality ℓ.
- **accept**(s) accept the establishing of a communication channel with the node having the site s. All authorized nodes have to be known in advance.
- **logout**(ℓ) removes the link between the action performing node and the node with locality ℓ.

They were introduced to make KLAIM suitable for dealing with open systems when *naming* might not be enough for establishing new connections [3]. This could be the case when networks routes are affected by restrictions, for example through firewall policies. Still, there is a limitation resulting out of the definitions of these actions. The syntax of the privileged action *accept* shows that all future connection partners of one node have to be known at design level. This limits the capability of KLAIM to handle the addition of completly unexpected nodes restricting the evolvement possibilities of the SOSANET.
A further mechanism for access control in KLAIM is a capability-based type system. It provides information about permissions for the download and consumption of tuples, the activation of processes and the creation of new nodes (more detailed information can be found in [7]).
Additionally KLAIM provides a programming language X-KLAIM (eXtended KLAIM)[3] with a high-level syntax for processes providing variable declarations, enriched operations, assignments, conditionals, sequential and iterative process composition. The implementation is based on KLAVA, a Java package providing the run-time system for X-KLAIM operations and a compiler for the X-KLAIM to Java translation. The hierarchical model of KLAIM is preserved

providing all primitives for the handling of dynamic node connectivity.

We introduced SOSANETs in this section describing its advantages over classic SANET designs. Furthermore, we presented the π-calculus and KLAIM pointing out its benefits for our goal and the limitations. In the following sections we will apply the described actions and processes of KLAIM to model service-oriented sensor-actuator networks.

4 A mobile SOSANET: The Driver Assistant System

We are working on a use case to evaluate the modeling power of our chosen π-calculus variant building a test surrounding for future work on network behavior analysis and verification. Our case study is a driver assistant system. It is illustrated in Figure 1.

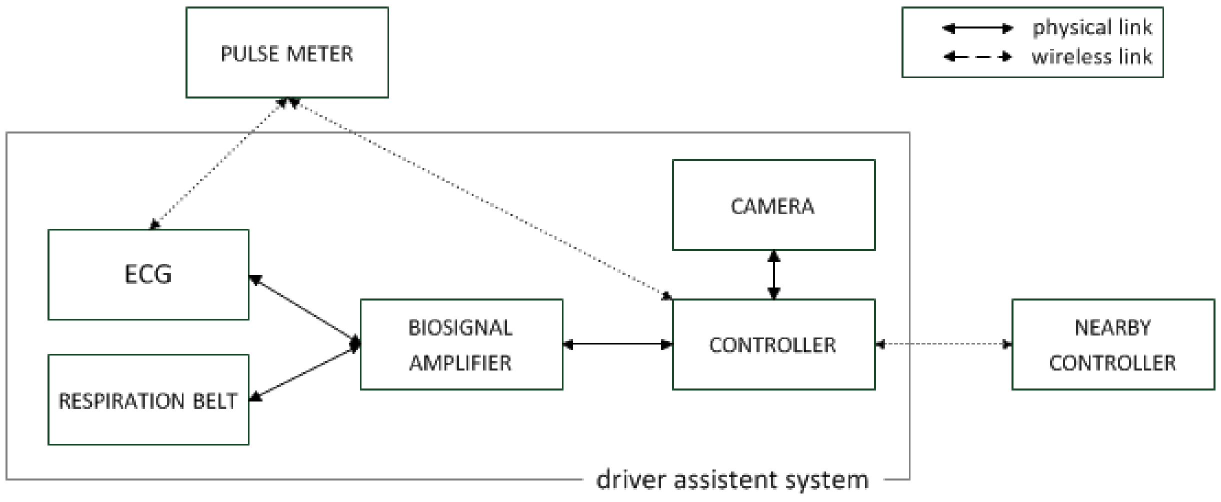

Figure 1: Use case scenario: a driver assistant system.

The system measures different vital signs of the driver and is able to react to unexpected events like heart attacks by sending an alarm and performing an emergency brake. The inner network is composed out of five wired components: a controller, and signal amplifier, a camera, a respiration belt and an electrocardiogram (ECG). The camera observes the head and eye movement, ECG and the respiration belt measure the breathing and the heart beat of the driver. The data evaluation and the activation of an alarm are part of the controller functionality. We assume that the driver also has a assistant system at his apartment securing his health status at home. This system requires a constant application of a portable pulse meter. It becomes part of the driver assistant network when the bearer enters the car by building wireless communication channels with the ECG and the controller. The data of the pulse meter becomes redundant when the ECG starts its measurements. In this case, the pulse meter can switch into a battery saving mode. One or more additional communication links can be established on-line in case of an emergency that requires information exchange with surrounding traffic participants. While the addition of the pulse meter node is an expected one, the later ones are not.

We focus in our modeling examples for clarity on four nodes and their interaction: the controller unit (cu), the amplifier (ampl) the ECG (ecg) and the pulse meter (pm) as illustrated in Figure 2. Hence, we have:

$$\begin{aligned} N \;=\;& s_{cu} ::^{\{s_{ampl},s_{pm}\}}_{s_{cu}/self,s_{ampl}/\ell_{ampl},s_{pm}/\ell_{pm}} P_{cu} \;||\; s_{ampl} ::^{\{s_{cu},s_{ecg}\}}_{s_{ampl}/self,s_{cu}/\ell_{cu},s_{ecg}/\ell_{ecg}} P_{ampl} \;|| \\ & s_{ecg} ::^{\{s_{ampl},s_{pm}\}}_{s_{ecg}/self,s_{ampl}/\ell_{ampl},s_{pm}/\ell_{pm}} P_{ecg} \;||\; s_{pm} ::^{\{s_{cu},s_{ecg}\}}_{s_{pm}/self,s_{cu}/\ell_{cu},s_{ecg}/\ell_{ecg}} P_{pm} \end{aligned}$$

This network description states that the network under observation consists out of four nodes which have the unique network identifiers (sites) s_{cu}, s_{ecg}, s_{ampl} and s_{pm}. The controller node is connected with the amplifier node and the pulse meter node mapping the respective sites with the associated localities. The set P_{cu} contains all processes which run concurrently on the node. The formula modeling the other nodes can be interpreted equivalently.

Modeling Topology Changes

We have to differentiate between four kinds of possible topology changes that can occur in a SOSANET.

The first one is the creation of a *new link between two familiar network nodes.* That could be the establishment of a new communication channel between already existing members of the network. Or it is the connection between a node which enters the network and a member of the network that is expecting the new node. This behavior can be modeled using the privileged NodeCoordinator actions login(ℓ) and accept(s) in combination with newloc(s,$\mathbb{C}$).

$$\begin{aligned} pm_{new} &= newloc(s_pm, \mathbb{C}_{pm}).\mathbb{C}'_{pm} \mid pm_{dorm} \\ \mathbb{C}'_{pm} &= login(\ell_{cu}).login(\ell_{ecg}).C''_{pm} \\ \mathbb{C}_{cont} &= \sum_{s_i} accept(s_i).\mathbb{C}_{cont} + \dots \\ \mathbb{C}_{ecg} &= \sum_{s_i} accept(s_i).\mathbb{C}_{ecg} + \dots \\ \dots & \end{aligned}$$

In the example above the entering of the pulse meter into the driver assistant network is described. First a new node with the network identifier s_{pm} is created and a NodeCoordinator instance is installed. The behavior of the node splits into two parallel components. One is the pulse meter in energy saving mode and the other a NodeCoordinator process. The NodeCoordinator tries to built new communication channels between the pulse meter and the ECG and between the pulse meter and the controller node. The corresponding NodeCoordinator processes have to synchronize to create the desired link. Furthermore, the site of pm has to be known by the accepting partner to complete the task.

Another possibility is that a *node with an unfamiliar site tries to enter the network.* This would be for example a controller node which belongs to a network installed in another car. The node wants to inform the first system that an emergency occurred in its driver cabin and that it is going to execute an emergency brake within the next moments. There is no possibility in KLAIM to model this scenario. We are only able to describe the creation of the new node and its attempts to open communication channels. An *accept* can not be processed due to the unknown site of the connecting node.

There is also the possibility to model the *controlled removal of a communication link* between two nodes.

$$\begin{array}{rcl} \mathbb{C}_{pm} & = & \mathit{logout}(\ell_{cu}).\mathit{logout}(\ell_{ecg}).\mathbb{C}_{pm} \mid \ldots \\ \ldots & & \end{array}$$

The pulse meter leaves the network by removing the previously established links. This happens when the driver is leaving the car after he reached his destination.

The last kind of topology changes we like to consider is the *spontaneous linkage failure.* These failures are only discoverable using special algorithms or test signals which check the functionality of the observed channel. The actions we used above are not appliable to model this behavior.

Modeling Service Functionalities

There are three basic operations in service oriented architectures needed for the interaction between service providers and service users: *service publishing, service discovery* and *service request.* In general, the *service publishing* function publishes the description of a service, its location and the associated service identifier into a central service registry. There is no such central registry in service-oriented sensor-actuator networks due to limited resources on each node. Furthermore, the ability of the network to change its topology ad hoc could lead to the disconnection of the central ”‘registry node”’ immobilizing the whole network.
Here, each node has a local registry storing informations about its own services and routes to other network services. This is implemented by writing 3-tuples of the form (”description”, id, locality) into the local tuple space. If the third tuple entry is **self**, it states, the service is installed on the node itself. Every other entry refers to a connected, neighboring node which has further information on the service locality in its tuple space. Figure 2 illustrates a service view on the network. Only a selection of services is published to keep the figure clear.

A formal notation of this function, here we observe the publishing of the *measure pulse* service, can be seen below. We focus on the exact execution order of each involved action and process omitting the description of possible process behavior alternatives for clarity reasons.

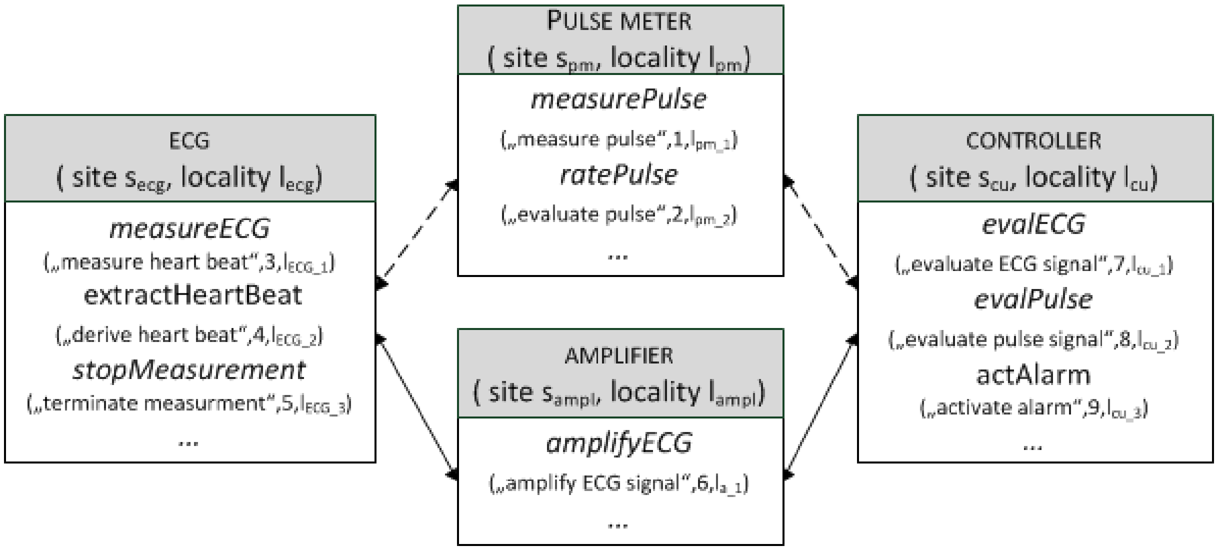
PULSE METER
(site s_pm, locality l_pm)
measurePulse
(„measure pulse",1,l_pm_1)
ratePulse
(„evaluate pulse",2,l_pm_2)
...
ECG
(site s_ecg, locality l_ecg)
measureECG
(„measure heart beat",3,l_ECG_1)
extractHeartBeat
(„derive heart beat",4,l_ECG_2)
stopMeasurement
(„terminate measurment",5,l_ECG_3)
...
CONTROLLER
(site s_cu, locality l_cu)
evalECG
(„evaluate ECG signal",7,l_cu_1)
evalPulse
(„evaluate pulse signal",8,l_cu_2)
actAlarm
(„activate alarm",9,l_cu_3)
...
AMPLIFIER
(site s_ampl, locality l_ampl)
amplifyECG
(„amplify ECG signal",6,l_a_1)
...

Figure 2: The reduced use case with a selection of offered services displayed at each node.

$$\begin{aligned}
pm &= out(\text{''}measure\ pulse\text{''}, 1, self)@self.out(\text{''}measure\ pulse\text{''}, 1, \ell_{pm})@\ell_{ECG}.\\
&\quad out(\text{''}measure\ pulse\text{''}, 1, \ell_{pm})@\ell_{cu}.pm\\
ECG &= out(in(!description, !service_{id}, !loc))@self.\\
&\quad out(description, service_{id}, loc)@\ell_{ampl}.ECG\\
ampl &= ...\\
...
\end{aligned}$$

First, the pulse meter publishes the existence of the service by writing the new tuple ("measure pulse",1,self) into its tuple space. Afterwards it forwards the information to all directly connected nodes. Then it behaves itself again like before.
The ECG node waits for invocation, receives the information and stores it in its own local tuple space before sending the received data to all neighboring nodes. This behavior is executed at each node in the network. The publishing process ends when a node receives a tuple which matches an existing one (not described above).

The second SOA operation, the *service discovery*, searches for an available service within the network. The search key is either the service id or a service description. The first one is used, if we are looking for a specific service. If we are satisfied with any service providing the needed functionality, we use a the second search key option. The search is done using the mobile process *discover*.
First, the controller node asks for the execution of the process *discover*. The process travels after invocation dynamically between nodes searching through the local tuple spaces for the locality of a service, which can measure the pulse (search key: "measure pulse"). Here the key and the locally saved tuples are compared with pattern matching mechanisms. The controller node splits

its behavior into two components, cu_1 waiting for a response and cu_3 proceeding. This decouples other functionalities of the node from the search activities avoiding a blockage of all available processes.

$$\begin{aligned} cu &= newloc(\ell_1).eval(discover(\text{"}measure\ pulse\text{"}, id, \ell_1))@\ell_{cu}. \\ &\quad ((in(!loc)@\ell_1.cu_1) \mid cu_3) \\ &\ldots \end{aligned}$$

Discover can match two tuples. The first one captures the locality information of the inquired service, back-propagates it through the network using the locality ℓ_1 and terminates the process *discover*. The other tuple captures the locality where the search has to be repeated.

$$\begin{aligned} discover(description, id, loc) &= read(\text{"}measure\ pulse\text{"}, id, !loc)@self.out(loc)@\ell_1.nil \\ &+ read(\text{"}measure\ pulse\text{"}, id, \ell_1')@self. \\ &\quad eval(discover(\text{"}measure\ pulse\text{"}, id, \ell_1))@\ell_1'.nil \\ &\ldots \end{aligned}$$

The execution of a services can be requested after discovering the service location. This in done using the *service request* operation. In our example the controller node wants to invoke a measurement of the pulse of the driver to check his health status. This means it requests an execution of the *measurePulse* service on the pulse meter node.
The instance of the controller that waited for the location of the queried service, sends a service request using the inquired locality information. Then it waits for the a transmission of the requested data. It behaves like cu_2 after receiving the data. The pulse meter node waits simultaneously for invocation. After receiving the request it splits its behavior into two component. One instance executes the requested service and terminates after sending the inquired data, the other behaves as before.

$$\begin{aligned} cu_1 &= newloc(\ell_2).out(\text{"}measurePulse\text{"}, \ell_2)@loc.in(!pulseData)@\ell_2.cu_2 \\ pm &= in(!reqService, !loc). \\ &\quad (< reqService > .out(pulseData)@loc.nil|pm) \\ cu_2 &= < evalPulse > .out(result)@self.cu_3 \\ cu_3 &= \ldots \\ \ldots \end{aligned}$$

The process cu_2 uses its own functionality *evalPulse* to evaluate the received data and to forward the result to process cu_3 which reacts appropriately (not noted).

As shown in this section, the given primitives and actions of KLAIM are expressive enough to formally describe features and behavior of SOSANETs. It is possible to formalize single node and complete network structures including available processes, communication channels and knowledge about dependencies between sites and localities. We are able to model process behavior, topology changes (the addition of expected nodes or the establishment and removal of connections) and the three central operations of service oriented architectures.

5 Conclusion and Future Work

We have illustrated the need for the analysis and verification of the behavior of service-oriented sensor-actuator networks applied in the medical domain. We could show that our chosen π-calculus variant KLAIM is suitable for modeling SOSA-NETs under the restriction that all topology changes are known in advance at design time. Furthermore, we could see that the provided actions for tuple manipulation are beneficial for the description of service-oriented routines. We have evaluated the applicability of KLAIM as formal modeling language for SOSANETs by modeling a driver assistant system. Furthermore, we identified the need to extend the language. Actions are needed, which are able to represent the addition of unexpected nodes. Without that, the interaction of unrelated networks cannot be analyzed limiting the constructional power of SOSANETs.
Our future work will focus on the described extension and on methods for the analysis and verification of network behavior. After finishing these goals we will be able to present a formal design framework for service-oriented sensor-actuator networks.

Acknowledgment. We thank the Geriatric Center of the Charité Berlin for familiarizing us with a closed driver assistant system which builds the basis of our case study.

References

[1] Ian F. Akyildiz & Ismail H. Kasimoglu (2004): *Wireless sensor and actor networks: research challenges*. *Ad Hoc Networks* 2(4), pp. 351 – 367, doi:10.1016/j.adhoc.2004.04.003. Available at `http://www.sciencedirect.com/science/article/pii/S1570870504000319`.

[2] D. Balasubramaniam, A. Dearle & R. Morrison (2008): *A composition-based approach to the construction and dynamic reconfiguration of wireless sensor network applications*. In: *Software Composition*, Springer, pp. 206–214, doi:10.1007/978-3-540-78789-1_16.

[3] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto & Betti Venneri (2003): *The Klaim Project: Theory and Practice*. *Global Computing: Programming Environments, Languages, Security and Analysis of Systems* 2874, pp. 88–150, doi:10.1007/978-3-540-40042-4_4.

[4] Doina Bucur & Marta Kwiatkowska (2011): *On software verification for sensor nodes*. *Journal of Systems and Software* 84(10), pp. 1693 – 1707, doi:10.1016/j.jss.2011.04.054. Available at `http://www.sciencedirect.com/science/article/pii/S0164121211001051`.

[5] Nicholas Carriero & David Gelernter (1989): *Linda in context*. *Commun. ACM* 32(4), pp. 444–458, doi:10.1145/63334.63337.

[6] R. De Nicola, G.L. Ferrari & R. Pugliese (1998): *KLAIM: a kernel language for agents interaction and mobility*. *Software Engineering, IEEE Transactions on* 24(5), pp. 315 –330, doi:10.1109/32.685256.

[7] R. De Nicola, G.L. Ferrari & R. Pugliese (2000): *Programming Access Control: The Klaim Experience*. *CONCUR 2000ŮConcurrency Theory*, pp. 48–65.

[8] R. De Nicola, D. Gorla & R. Pugliese (2006): *On the expressive power of KLAIM-based calculi*. *Theoretical Computer Science* 356(3), pp. 387–421, doi:10.1016/j.tcs.2006.02.007.

[9] A. Dearle, D. Balasubramaniam, J. Lewis & R. Morrison (2008): *A component-based model and language for wireless sensor network applications.* In: *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, IEEE, pp. 1303–1308, doi:10.1109/COMPSAC.2008.151.

[10] M. Diaz, D. Garrido, L. Llopis, B. Rubio & J.M. Troya (2006): *A Component Framework for Wireless Sensor and Actor Networks.* In: *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pp. 300 –307, doi:10.1109/ETFA.2006.355382.

[11] A. Dunkels, B. Gronvall & T. Voigt (2004): *Contiki-a lightweight and flexible operating system for tiny networked sensors.* In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, IEEE, pp. 455–462, doi:10.1109/LCN.2004.38.

[12] Adrian Francalanza & Matthew Hennessy (2008): *A theory of system behaviour in the presence of node and link failure.* *Information and Computation* 206(6), pp. 711 – 759, doi:10.1016/j.ic.2007.12.002. Available at `http://www.sciencedirect.com/science/article/pii/S0890540108000023`.

[13] Paul Harvey, Alan Dearle, Jonathan Lewis & Joseph S. Sventek (2012): *Channel and Active Component Abstractions for WSN Programming - A Language Model with Operating System Support.* In Marten van Sinderen, Octavian Postolache & César Benavente-Peces, editors: *SENSORNETS*, SciTePress, pp. 35–44. Available at `http://dblp.uni-trier.de/db/conf/sensornets/sensornets2012.html#HarveyDLS12`.

[14] J. Khalil Jacoub, R. Liscano & J. Bradbury (2011): *A Survey of Modeling Techniques for Wireless Sensor Networks.* In: *SENSORCOMM 2011, The Fifth International Conference on Sensor Technologies and Applications*, pp. 103–109.

[15] J. Khalil Jacoub, R. Liscano & J. Bradbury (2012): *Assessment of Software Modeling Techniques for Wireless Sensor Networks: A Survey.* *Sensors and Transducers* 14-2, pp. 18–46.

[16] Alan M. Mainwaring, David E. Culler, Joseph Polastre, Robert Szewczyk & John Anderson (2002): *Wireless sensor networks for habitat monitoring.* In: *WSNA*, pp. 88–97. Available at `http://doi.acm.org/10.1145/570738.570751`.

[17] Diego Martinez, Apolinar Gonzalez, Francisco Blanes, Raul Aquino, Jose Simo & Alfons Crespo (2011): *Formal Specification and Design Techniques for Wireless Sensor and Actuator Networks.* *Sensors* 11(1), pp. 1059–1077, doi:10.3390/s110101059. Available at `http://www.mdpi.com/1424-8220/11/1/1059`.

[18] Elena Meshkova, Janne Riihijärvi, Frank Oldewurtel, Christine Jardak & Petri Mähönen (2008): *Service-Oriented Design Methodology for Wireless Sensor Networks: A View through Case Studies.* In: *SUTC*, pp. 146–153. Available at `http://doi.ieeecomputersociety.org/10.1109/SUTC.2008.43`.

[19] R. Milner (1989): *Communication and concurrency.* Prentice-Hall, Inc.

[20] Robin Milner (1999): *Communicating and Mobile Systems: The pi-Calculus.* Cambridge University Press.

[21] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, I.* *Inf. Comput.* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[22] V.A. Oleshchuk (2003): *Ad-hoc sensor networks: modeling, specification and verification.* In: *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings of the Second IEEE International Workshop on*, IEEE, pp. 76–79, doi:10.1109/IDAACS.2003.1249521.

[23] M.P. Papazoglou (2003): *Service-oriented computing: concepts, characteristics and directions.* In: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pp. 3 – 12, doi:10.1109/WISE.2003.1254461.

[24] M. Patrignani, N. Matthys, J. Proenca, D. Hughes & D. Clarke (2012): *Formal analysis of policies in wireless sensor network applications.* In: *Software Engineering for Sensor Network Applications (SESENA), 2012 Third International Workshop on*, pp. 15 –21, doi:10.1109/SESENA.2012.6225728.

[25] Abdelmounaam Rezgui & Mohamed Eltoweissy (2007): *Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead.* *Computer Communications* 30(13), pp. 2627–2648. Available at `http://dx.doi.org/10.1016/j.comcom.2007.05.036`.

[26] O. Sharma, J. Lewis, A. Miller, A. Dearle, D. Balasubramaniam, R. Morrison & J. Sventek (2009): *Towards verifying correctness of wireless sensor network applications using Insense and SPIN.* *Model Checking Software* 5578, pp. 223–240, doi:10.1007/978-3-642-02652-2_19.