K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

The π-Calculus for SoS: Novel π-Calculus for the Formal Modeling of Software-intensive Systems-of-Systems

Flavio OQUENDO[1]
IRISA – UMR CNRS, Univ. de Bretagne-Sud, France

Abstract. A major research challenge in the architectural design of a software-intensive System-of-Systems (SoS) is to enable the formal modeling of its evolutionary architecture. One of the main issues is that SoS architectures evolve dynamically, during run-time, in unexpected ways while producing emergent behavior. To address this issue, this paper proposes a novel process calculus, called "the π-Calculus for SoS", defined as a novel variant of the π-Calculus based on concurrent constraints and inferred channel bindings for enabling the formal modeling of software-intensive SoSs, meeting their challenging architectural characteristics.

Keywords. π-calculus, concurrent constraints, formal modeling languages, software architecture, software-intensive systems-of-systems.

Introduction

The pervasiveness of communication networks has made increasingly possible to interconnect software-intensive systems that are developed, operated, managed, and evolved independently. The result is a new kind of complex software-intensive system, the so-called System-of-Systems (SoS), which develops evolutionarily by composing several existing systems to achieve missions not possible with a single system.

This is the case of SoSs found in different areas as diverse as aeronautics, energy, healthcare, manufacturing, and transportation; and applications that address societal needs as e.g. environmental monitoring, emergency coordination, traffic control, smart grids, and smart cities. Moreover, enabling platforms such as the Internet of Things (generalizing wireless sensor/actuator networks in the Cloud) and nascent classes of SoSs such as cyber-physical ones are accelerating the deployment of software-intensive SoSs [28].

An SoS has intrinsic characteristics that are hard to address when compared to those of single systems [19]. In an SoS, constituent systems are: operationally independent, managerially independent, geographically distributed and physically decoupled (limiting the exchange to information only). The SoS as a whole is always subject to evolutionary development and emergent behavior, i.e. new behavior that stem from the local interactions among constituent systems, but cannot be deduced from the behaviors of the constituent systems themselves.

Complexity is intrinsically associated to emergent behavior; a complex system being defined to be a system of interacting parts that displays emergent behavior. By its nature, by

[1] Corresponding author: *Flavio Oquendo, IRISA – UMR CNRS, Univ. de Bretagne-Sud, France*; E-mail: `flavio.oquendo@irisa.fr`

definition [19], SoSs always produce emergent behaviors. In SoSs, missions are achieved through emergent behaviors [36].

Hence, complexity poses the need for separation of concerns between architecture and engineering: architecture focuses on design and analysis about interactions of parts and their emergent properties while engineering focuses on designing and constructing such parts and integrating them as architected.

More specifically, according to ISO/IEC/IEEE Standard 42010 [15], an SoS architecture can be defined as the fundamental organization of an SoS embodied in its constituent systems, their relationships to each other and to the environment, and the principles guiding its design and evolution.

Conceiving Architecture Description Languages (ADLs) for software-intensive systems has been the subject of intensive research in the last two decades resulting in the definition of several ADLs for formally describing the architectures of (often large) single systems [21][22][30]. However, none of these ADLs has the expressive power to formally describe the architecture of software-intensive SoSs [12][18].

Indeed, all these ADLs for single systems have formal foundations that only cope with the specification of architectures that are static (i.e. architectures defined at design-time which never change at run-time) or dynamic (i.e. architectures which may change at run-time according to the anticipated reconfigurations known at design-time).

However, SoS architectures are evolutionary, being unpredictably dynamic due to emergent behaviors [11]. Moreover, the actual constituent systems of an SoS are generally not known at design-time and are only selected at run-time. Therefore, the intrinsic characteristics of SoSs lead to software architectures at run-time that cannot be predicted at design-time [19]. To address the research challenge raised by the unique characteristics of SoSs, the targeted breakthrough is to conceive a novel ADL providing the formal foundation to describe SoS architectures, which may change unpredictably at run-time in SoSs [11][12][18].

As a matter of fact, in ADLs for describing the architecture of single software-intensive systems, process calculi have been shown to constitute the suitable mathematical foundation. As expected, the main concern in the formal description of these architectures is how structure and behavior are interrelated in concurrent sets, and this is the purpose of process calculi as formal theory [37].

To fill this gap, we conceived a novel process calculus in the family of the π-Calculus [23]. Named "the π-Calculus for SoS", it provides the primary formal foundation having the expressive power to address the challenge of modeling the evolutionary architecture of software-intensive SoSs. Based on this foundation, we defined an ADL for SoS, named SosADL [26].

This paper focuses on the presentation of the π-Calculus for SoS from the communicating process architecture viewpoint. It complements three others recently published: [26] and [27] that presented the ADL for SoS, i.e. SosADL, and its formal foundation from the viewpoint of SoS architects at the IEEE SoS Engineering Conference – SoSE 2016; and [29] that described the validation of SosADL based on a real pilot project and related case study to be presented in the IEEE International Conference on Systems, Man, and Cybernetics – SMC 2016.

The remainder of this paper is organized as follows. Section 1 describes the motivation for conceiving a novel process calculus for SoS and presents related work. Section 2 presents the formal definition of the π-Calculus for SoS in terms of its abstract syntax and its structural operational semantics expressed in terms of labeled transition rules. In section 3, we describe how to apply the π-Calculus for SoS for modeling the architectural behaviors of an SoS. In section 4, we briefly introduce the supporting toolset and the validation of the π-Calculus for SoS. To conclude we summarize, in section 5, the main contributions of this paper and outline future work.

1. Motivation and Related Work

The combination of intrinsic SoS characteristics turns SoS architectures to be naturally highly evolvable, unpredictably changing at run-time with regard to their constituent systems and operational environment. SoSs have evolutionary architectures (with the meaning that they dynamically adapt and continuously evolve at run-time in ways not envisaged at design-time).

As mentioned, many works have addressed the issue of formally describing the architecture of single software-intensive systems [21][22][30]. Most of the work carried out addressed how existing or extended process calculi and related languages enable the formal description of software architectures.

Therefore, we must pose the question: do the process calculi constituting the formal foundations of these ADLs provide enough expressive power for modeling SoS architectures?

To answer this question, let us analyze the state-of-the-art on formal foundations of ADLs for single software-intensive systems. In fact, different process calculi were applied as formal foundations for describing the architecture of single systems, of which the main ones, from an ADL perspective, are: FSP [20] (the formal foundation of Darwin [17]), CSP [14] (the formal foundation of Wright [1]), and π-Calculus [23] (the formal foundation of π-ADL [6][25]). In addition, emerging formal languages for modeling contracts in SoSs [3], of which the main representative is CML [40], base their foundations on variants of process calculi, in particular CSP-CIRCUS for CML [41].

These process calculi have been applied to formalize architecture description where:

- a "component" is specified as a process interacting with its environment by providing ports specified by channel names and having its behavior specified in a specific process calculus;
- a "connection" is specified as a channel binding for linking together ports of different components, enabling communication between connected processes (besides simple connections, more complicated "connectors" may be specified).

Figure 1 below depicts these typical structural elements of process calculi in terms of a flow graph, where processes *P* and *Q* are composed, the former exposing channel names *a*, *complement of b*, and *c* for interaction and the later channel names *b*, *d*, and *e* and where also a channel binding is declared between *b* and its complement enabling interaction between *P* and *Q*.

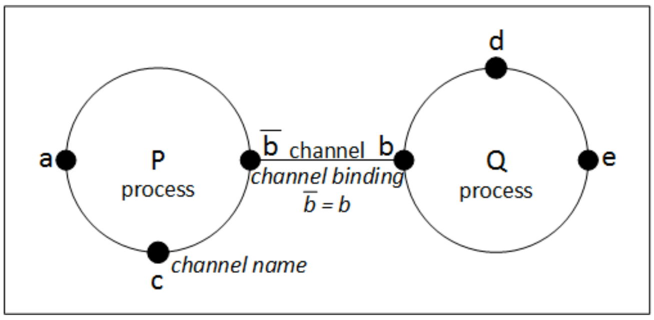


Figure 1: Flowgraph showing typical structural elements of process calculi.

The formal mechanisms for declaring processes, channel names and their bindings vary from a process calculus to another. In particular, regarding ADLs, the binding mechanism is a discriminating feature of different process calculi. Concerning process calculi grounding ADLs for single systems, the binding mechanism is shared actions in FSP/Darwin, statistic channels in CSP/Wright and CSP-CIRCUS/CML, and extruded channels in π-Calculus/π-ADL.

Four aspects of the binding mechanism are key for supporting different kinds of architecture description:

- endogenous vs exogenous bindings: are the channel bindings decided within the process definitions, i.e. endogenously, or outside in the definition of their compositions, i.e. exogenously?
- unconstrained vs constrained bindings: are the channel bindings unconstrained, i.e. once declared they always hold, or constrained, i.e. a channel binding will hold only if it satisfies a specific set of properties?
- extensional vs intentional bindings: are the channel bindings declared explicitly in the process composition or not, i.e. intentionally by a set of properties?
- unmediated vs mediated bindings: are the channel bindings created without any mediation between the related processes, i.e. unmediated, or is the channel binding created after mediation between related processes taking into account mutually agreed properties?

Regarding these four aspects of bindings in process calculi grounding ADLs for single systems, they are:

- endogenously decided at design-time;
- unconstrainedly specified;
- extensionally declared at design-time (even if π-Calculus supports mobility at run-time, while FSP and CSP[2] do not);
- unmediated between processes.

In particular, in FSP and CSP the channel bindings are defined explicitly, by extension, in the process definitions (design-time) and cannot change during their application (run-time). In the π-Calculus, channel bindings may evolve according to the interaction of processes based on channel mobility, however the linkage structure is completely defined in an endogenous way.

By the nature of single system architectures, these different process calculi capture the needs for establishing the formal foundations of single system ADLs. Indeed, architectures of single systems are basically static (where all the architectural decisions are taken at design/definition time) or dynamic (where all anticipated reconfigurations are known at design/definition-time).

Indeed, all ADLs for single systems have formal foundations that only cope with the specification of architectures that are static or dynamic by having their binding mechanism being endogenous, unconstrained, extensional, and unmediated.

Let us now come back to the posed question: do the process calculi constituting the formal foundations of these ADLs provide enough expressive power for modeling SoS architectures?

The answer is clearly: no, they do not. None of these process calculi provides a suitable basis for formalizing SoS architectures [11]. Necessarily, the description of SoS architectures needs to specify:

- a “constituent” as a process interacting with its local environment;
- a “connection” as a channel binding inferred from local environments, subject to

[2] Note also that variants of CSP have been defined with extensions enabling the mobility of channels inspired by the π-Calculus [38].

uncertain information, i.e. actual bindings must only be decided at run-time among actual constituents of the SoS, not at design-time like in single systems.

Moreover, "mediators" need to be defined in order to cope with constraints coming from processes to be linked by connections.

In the case of SoSs, the channel bindings connecting constituents together enabling interaction must thereby be:

- exogenous: interactions in an SoS are due to the mission of the SoS and not to the missions of the constituent systems themselves, which moreover are not necessarily aware of the SoS mission (interactions must therefore by exogenous to the constituents, and not endogenous like in single systems);
- constrained: interactions in an SoS are constrained by local environments (as opposed to single systems whose architectures are unaware of local contexts) and in particular subject to uncertain information;
- intentional: interactions in an SoS are actually decided at run-time (not extensionally decided at design-time like in single systems, in particular as concrete SoS constituents are often not known at design-time);
- mediated: interactions in an SoS must be mediated by local environments of constituent systems (unlike in single system architectures where decisions are made at design-time and are thereby unmediated).

As none of these process calculi meet the needs of SoSs, we must pose the subsequent question: beyond these process calculi underlying single system ADLs, are there other process calculi that are suitable for describing SoS architectures?

To answer this question, we have evaluated different process calculi developed for modeling complex systems, beyond engineered systems and systems-of-systems, and identified several forms of the π-Calculus that were developed for modeling natural complex systems in Biology [34] and Chemistry [32]. Some other process calculi were also extended with the mobility mechanisms of the π-Calculus, e.g. CSP with mobile channels [38] and its derived occam-π language which was used for modeling natural systems [35].

For answering that question, we further evaluated the π-Calculus in its original form [23] as well as its enhanced forms that have been developed along the years.

This evaluation included, on the one hand, general-purpose π-Calculi and, on the other hand, π-Calculi that were specially developed for modeling complex systems. The conclusion of this study is that the binding mechanism underlying interactions for creating composite behaviors in:

- the original π-Calculus [23] is endogenous, unconstrained, extensional, and unmediated;
- the Fusion-Calculus [31], extending the original π-Calculus, is exogenous, unconstrained, extensional, and unmediated;
- the explicit fusion π-F-Calculus [39], extending the Fusion-Calculus, is exogenous, constrained, extensional, and unmediated;
- the Attributed π-Calculus [16], extending the π-F-Calculus, is exogenous, constrained, extensional, and unmediated;
- the Constrained CC-π-Calculus [5], also extending the π-F-Calculus, is exogenous, constrained, extensional, and unmediated.

Again, the answer is: no, they do not. None of these process calculi provides a suitable basis for formalizing the architecture of SoSs. Therefore, a novel process calculus is needed as a suitable formal foundation of an ADL for SoS as none of existing π-Calculi meets SoS architecture needs.

The design decisions underlying the π-Calculus for SoS are: (i) processes must be constrained by their local environments; and (ii) bindings enabling interaction between processes must be exogenous, constrained, intentional, and mediated.

Our approach to design a novel π-Calculus coping with SoS needs was to generalize the original π-Calculus with mediated concurrent constraints, where mediation is achieved by constraining channel bindings (bindings will appear, disappear or reappear according to mediated constraints between concurrent processes).

Straightforwardly speaking, static process calculi such as FSP (underlying Darwin) and CSP (underlying Wright) specify a single system architecture in terms of "unconstrained processes and fixed channels", dynamic process calculi such as π-Calculus (underlying π-ADL) specify a single system architecture in terms of "unconstrained processes and mobile channels", while the π-Calculus for SoS specifies a system-of-systems architecture in terms of "local constrained processes and mediated constrained channels".

Therefore, for meeting the needs of SoS architecture description, the π-Calculus for SoS generalizes the π-Calculus with the notion of "concurrent constraints" based on the Concurrent Constraint Paradigm (CCP)[3] [24][10], and on the principles of CCP-based calculi [24]. More specifically, it enhances the CC-π-Calculus with constraint-mediated channel bindings while providing a more expressive constraint specification formalism.

CCP is based on four principles: (i) concurrency (processes which run concurrently); (ii) communication (these running processes interact according to the accumulation of constraints in a shared environment); (iii) coordination (the presence or absence of information in the shared environment guards execution of a running process); and (iv) localization (in addition to the shared environment, each running process has access only to local, varying pieces of information, and may create new pieces of information on the fly).

A fundamental concept in CCP is the specification of composition of concurrent processes by means of constraints. A constraint may represent partial information on the state of the environment. During the computation, the current state of the environment is specified by a set of constraints. Processes can change the state of the environment by telling information (i.e. adding constraints), and can synchronize by asking information from the environment (i.e. determining whether a given constraint can be inferred from the told constraints).

Typically, CCP calculi contain the following primitives [24]: (i) a *tell* operator for adding a constraint to the shared environment; (ii) an *ask* operator for inquiring if a constraint can be inferred from the shared environment; (iii) parallel composition combining processes concurrently; (iv) a restriction operator introducing local variables, restricting the interface that the process can use to interact with others.

Intuitively speaking, using CCP primitives, a process can publicly 'tell' the shared environment about pieces of information that it knows, while maintaining private information internally. A process can also 'ask' information from the shared environment which influences on its own behavior.

It is also worth noting that 'telling' and 'asking' constraints support computation with partial information. Thereby, another noteworthy difference of the formal basis of ADL for single systems (e.g. FSP, CSP and π-Calculus) and the π-Calculus for SoS is the treatment of partial information (i.e. each process has incomplete information on the state of the environment as a whole, this information being limited to its local environment). While in single systems, all components may have access to complete information, in SoS, all constituents have access to incomplete information by nature. In SoSs, partial information contributes to uncertainty, in addition to the uncertainty of emergent behavior.

Summing up, the π-Calculus for SoS extends the π-Calculus with CCP primitives and

[3] Including calculi such as Concurrent Constraint Programming and Concurrent Constraint Logic Programing.

inferred channel bindings from mediated concurrent constraints embodying SoS characteristics. It subsumes the original π-Calculus as well as extensions based on fusions, explicit fusions, attributes, and purely constraints.

2. Formal Definition of the π-Calculus for SoS

The formal definition of the π-Calculus for SoS encompasses its formal abstract syntax and formal semantics. The basis for defining its formal semantics is the defined abstract syntax[4] shown in Figure 2.

The formal semantics of the π-Calculus for SoS is defined by means of a formal transition system, expressed by labeled transition rules, which are formulated as proof rules.

In a transition rule, shown in Figure 3, premises and conclusions are transition relations. Thereby, if the transition relations labeled by $\alpha_1 \ldots \alpha_n$ can fire, then the transition relation labeled by α can fire, i.e. if P_1 can fire α_1 and become P_1' … and P_n can fire α_n and become P_n', then C can fire α and become C'. Note that $P_1 \ldots P_n$ are terms of C and $P_1' \ldots P_n'$ are terms of C'. Side conditions can also be expressed. They are necessary conditions on terms expressed in the premises.

Abstract syntax of π-Calculus for SoS

constrainedBehavior ::= behavior$_1$
  | restriction$_1$ **.** constrainedBehavior$_1$ -- *Constrained Behavior*
  | **behavior** name$_1$ **(** value$_0$ …**,** value$_n$ **) is {** behavior$_1$ **}** -- *Definition*
  | **constraint** name$_1$ **is {** constraint$_1$ **}** -- *Constraint Definition*
  | **compose {** constrainedBehavior$_0$ … **and** constrainedBehavior$_n$ **}**
behavior ::= baseBehavior$_1$
  | restriction$_1$ **.** behavior$_1$ -- *Unconstrained Behavior*
  | **repeat {** behavior$_1$ **}** -- *Repeat*
  | **apply** name$_1$ **(** value$_0$ …**,** value$_n$ **)** -- *Application*
  | **compose {** behavior$_0$ … **and** behavior$_n$ **}** -- *Composition*
baseBehavior ::= action$_1$ **.** behavior$_1$ -- *Sequence*
  | **choose {** action$_0$ **.** baseBehavior$_0$ -- *Choice*
  or action$_1$ **.** baseBehavior$_1$ … **or** action$_n$ **.** baseBehavior$_n$ **}**
  | **if** constraint$_1$ **then {** baseBehavior$_1$ **} else {** baseBehavior$_2$ **}**
  | **done** -- *Termination*
action ::= baseAction$_1$
  | **tell** constraint$_1$ -- *Tell*
  | **untell** constraint$_1$ -- *Unsaid*
  | **check** constraint$_1$ -- *Check*
  | **ask** constraint$_1$ -- *Ask*
baseAction ::= **via** connection$_1$ **send** value$_0$ -- *Output*
  | **via** connection$_1$ **receive** name$_0$ **:** type$_0$ -- *Input*
  | **unobservable** -- *Unobservable*
connection ::= **connection** name$_1$
restriction ::= **value** name$_1$ **=** value$_0$ | connection$_1$

Figure 2: Abstract syntax of typed behaviors and values.

[4] The abstract syntax of the π-Calculus for SoS is defined using the following notation for the abstract production rules: (i) keywords are written with bold; (ii) non-terminals are written without bold; (iii) a sequence of zero, one or more elements is written: $Element_{min}$, …, $Element_{max}$, where the value of *min* specifies the minimum number of elements (*0* specifies possibly no elements, *1* specifies at least one element) and the value of *max* specifies the maximum number of elements ($Element_n$ specifies any number of elements); (iv) alternative choices are written separated by |.

$$\textbf{\textit{Transition rule:}}\quad \frac{P_1 \xrightarrow{\alpha_1} P_1' \ \ldots\ P_n \xrightarrow{\alpha_n} P_n'}{C \xrightarrow{\alpha} C'}$$
$$\textbf{where}\ \textit{side conditions}$$

Figure 3: Form of a labeled transition rule in terms of proof rule.

Using transition rules, the π-Calculus for SoS is defined in terms of structural operational semantics [2]: first we define transition rules (axioms) for actions (shown in Figure 4) and then define the transition rules representing behaviors (shown in Figure 5) composed by these actions and defined behaviors[5].

2.1 Structural Operational Semantics for Actions

In this formal definition, every construct of the π-Calculus for SoS expressed in its abstract syntax has its structural operational semantics specified by a set of transition rules (as seen in Figure 4 for actions and in Figure 5 for behaviors performing these actions). Each transition rule specifies a possible behavior associated to a construct that straightforwardly corresponds to the defined abstract syntax.

Precisely, regarding the formal semantics of actions shown in Figure 4, the first three labeled transition rules, i.e. *Output*, *Input*, and *Unobservable* define the semantics of the π-Calculus for SoS constructs expressing respectively the output action ***via*** *connection*$_1$ ***send*** *value*$_1$, the input action ***via*** *connection*$_1$ ***receive*** *value*, and the internal action ***unobservable***. It means that in these three cases, we have three axioms that can always apply for firing atomic behaviors.

In the first case, executing action ***via*** *connection*$_1$ ***send*** *value*$_1$ implies that *value*$_1$ is sent through *connection*$_1$ and the continuation of the behavior is the sequel *behavior*$_1$. In the second case, executing action ***via*** *connection*$_1$ ***receive*** *value* implies that a *value*$_1$ is received through *connection*$_1$ and the continuation of the behavior is the sequel *behavior*$_1$ where the variable *value* is bound to the received *value*$_1$ by adding an equality constraint (*value* = *value*$_1$) to the local environment. In the third case, executing any internal, ***unobservable*** action implies that the continuation of the behavior is the sequel *behavior*$_1$. Note that the *Output* and *Input* are complementary transition rules that need to be executed together, in synchronization, as defined by the *Communication* transition rule (shown in Figure 5).

The next transition rules, *Tell* and *Untell*, define constraint-handling constructs that are grounded on constraint satisfaction (see [13] for details). The former, i.e. *Tell*, adds a new constraint to the local environment (if and only if constraints together with the new told constraint are consistent[6]). The later, i.e. *Untell*, removes a constraint from the local environment, if it exists, while maintaining its consistency.

The following transition rule, i.e. *Check*, checks if a constraint and the set of constraints in the local environment would be consistent all together. After execution, the behavior continues by executing the sequel *behavior*$_1$. If not, it remains blocked. This construct therefore supports checking the consistency of a new constraint with the local environment before possibly telling it.

The next rule, i.e. *Ask*, asks whether a constraint can be derived from the constraints in the local environment. If yes, i.e. this constraint can be entailed, it continues by executing the continuation of the behavior, i.e. the sequel *behavior*$_1$. If not, it remains blocked.

[5] The term behavior is used instead of process in π-Calculus for SoS.

[6] In the sense of logical consistency: a consistent set of constraints is one that does not contain a contradiction.

Formal semantics of π-Calculus for SoS: labeled transition rules for actions

***Output*:**

$$\frac{}{\textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\,(\textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{value}_1\,.\,\text{behavior}_1)\end{array}\right\} \xrightarrow{\textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{value}_1} \textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ \text{behavior}_1\right\}}$$

***Input*:**

$$\frac{}{\textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\,(\textbf{via}\ \text{connection}_1\ \textbf{receive}\ \text{value}\,.\,\text{behavior}_1)\end{array}\right\} \xrightarrow{\textbf{via}\ \text{connection}_1\ \textbf{receive}\ \text{value}_1} \textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\,(\text{value} = \text{value}_1)\\ \textbf{and}\ \text{behavior}_1\end{array}\right\}}$$

where $(\text{constraint}_{0..n}$ **and** $(\text{value} = \text{value}_1))$ *is consistent, i.e. binding* $(\text{value} = \text{value}_1)$ *can be consistently asserted together with* $\text{constraint}_{0..n}$

***Unobservable*:**

$$\frac{}{\textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ (\textbf{unobservable}\,.\,\text{behavior}_1)\right\} \xrightarrow{\tau} \textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ \text{behavior}_1\right\}}$$

***Tell*:**

$$\frac{}{\textbf{compose}\left\{\text{constraint}_{0..m}\ \textbf{and}\ (\textbf{tell}\ \text{constraint}_n\,.\,\text{behavior}_1)\right\} \xrightarrow{\tau} \textbf{compose}\left\{\text{constraint}_{0..m}\ \textbf{and}\ \text{constraint}_n\ \textbf{and}\ \text{behavior}_1\right\}}$$

where $(\text{constraint}_{0..m}$ **and** $\text{constraint}_n)$ *is consistent, i.e.* constraint_n *can be consistently asserted with* $\text{constraint}_{0..m}$

***Untell*:**

$$\frac{}{\textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ (\textbf{untell}\ \text{constraint}_m\,.\,\text{behavior}_1)\right\} \xrightarrow{\tau} \textbf{compose}\left\{(\text{constraint}_{0..n} - \text{constraint}_m)\ \textbf{and}\ \text{behavior}_1\right\}}$$

where $(\text{constraint}_{0..n} - \text{constraint}_m)$ *is consistent, i.e.* constraint_m *can be consistently retracted from* $\text{constraint}_{0..n}$

***Check*:**

$$\frac{}{\textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ (\textbf{check}\ \text{constraint}_m\,.\,\text{behavior}_1)\right\} \xrightarrow{\tau} \textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ \text{behavior}_1\right\}}$$

where $(\text{constraint}_{0..n}$ **and** $\text{constraint}_m)$ *is consistent, i.e.* constraint_m *is checked to be consistent with* $\text{constraint}_{0..n}$

***Ask*:**

$$\frac{}{\textbf{compose}\left\{\text{constraint}_{0..m}\ \textbf{and}\ (\textbf{ask}\ \text{constraint}_n\,.\,\text{behavior}_1)\right\} \xrightarrow{\tau} \textbf{compose}\left\{\text{constraint}_{0..m}\ \textbf{and}\ \text{behavior}_1\right\}}$$

where $\text{constraint}_{0..m} \mid\!- \text{constraint}_n$, *i.e.* constraint_n *can be derived from* $\text{constraint}_{0..m}$

Figure 4: Formal semantics of actions in the π-Calculus for SoS.

2.2 Structural Operational Semantics for Behaviors

Let us now present the labeled transition rules defining the formal semantics of behaviors, shown in in Figure 5.

The transition rule *Restriction* defines the semantics of local declaration of connection restricting scope. It declares that a connection that is restricted to a behavior has no impact on other actions carried out by the behavior.

The transition rule *Communication* defines the semantics of communication between behaviors, where one is ready to send and the other ready to receive via a bound connection. Once the communication is fired, a new constraint is added to the environment telling that the variable *value* is equal to the sent *value*$_1$.

The restriction within communication is formalized by the transition rule *Restriction-Open*, where a restricted value is passed to another behavior that may already have this value in its scope or not. In the former situation, its scope is not changed. In the latter situation, its scope is extended to the other behavior (a.k.a. scope extrusion, the resulting scope embracing both behaviors).

Expression of scope extrusion results from the transition rule *Restriction-Open* followed by the transition rule *Communication-Close.*

The transition rule *Communication-Close* defines that if a behavior passes a restricted value to another, the scope of this value is extruded to include both the behavior that sent it and the behavior that received it. This is particularly interesting in the case the value sent is a connection, which expresses the extrusion of local connections.

Note that the side condition in this case inhibits a behavior from communicating through a connection that was declared as restricted. It is worth noting also that its sub-behaviors can use this restricted connection to communicate between them (even if the behavior itself cannot).

Note also that transition rules *Restriction-Open* and *Communication-Close* define together the expression of scope extrusion where the rule *Restriction-Open* enables a value to be opened to a new scope and then the rule *Communication-Close* closes this new scope to embrace both behaviors (the one that sent the restricted connection and the one that received it). Scope extrusion thereby supports dynamic reconfiguration.

The transition rule *Choice* defines the semantics of the choice construct ***choose*** { $action_0 . behavior_0$ … ***or*** $action_m . behavior_m$ }. This rule can be triggered when the prefix action is enabled in any of the behaviors. In case several prefix actions are enabled, the choice is non-deterministic.

Note that there is no transition rule for inferring transitions of ***done***, as the inaction represents that no transition is possible (it is the equivalent of a choose with no choices).

The conditional construct ***if*** *constraint* ***then*** $behavior_1$ ***else*** $behavior_2$ is defined by the next two rules, i.e. *Conditional-Then* and *Conditional-Else.*

The side condition $constraint \equiv true$ or $constraint \equiv false$ implies that it will be $behavior_1$ (the *then* behavior) or it will be $behavior_2$ (the *else* behavior) that is enabled to be executed according to the truth value.

The iteration construct ***repeat*** { $behavior_1$ } is defined by the transition rule *Repetition* enabling repeated execution of $behavior_1$.

The construct ***compose*** { $constrainedBehavior_0$ … ***and*** $constrainedBehavior_n$ } is defined by the transition rule *Composition*. This rule means that composition may trigger the execution of any of its behaviors concurrently according to the told constraints in its environment. It is worth noting that this construct encompasses the case of behaviors subject to no constraint: the construct ***compose*** { $constrainedBehavior_0$ … ***and*** $constrainedBehavior_n$ } encompasses the construct ***compose*** { $behavior_0$ … ***and*** $behavior_n$ } which is a syntactical restriction of the former where no constraints are specified.

2.3 Constraint System for Handling Constraints in the π-Calculus for SoS

Let us now complete the formal definition of the π-Calculus for SoS with the definition of the formal constraint system. We defined the π-Calculus for SoS parameterized by a call-by-value data expression formalism [4] providing means to compute values as well as to impose constraints on interactions.

A constraint system provides a signature, defining a constraint language, from which the constraints can be expressed and an entailment relation (denoted □) for specifying interdependencies between constraints. A constraint represents a piece of information (or partial information) upon which processes may act. The inter-dependency, *c1* □*c2*, expresses that the information specified by *c2* can be entailed from the information specified by *c1*.

Let Σ be a signature (i.e. a set of constant, function and predicate symbols with their arity and typing) and Δ be a consistent first-order theory over Σ (i.e. a set of sentences over Σ having at least one model).

Constraints are first-order formulae over Σ. We can then state that *c1* □*c2* if the implication *c1* □ *c2* is valid in Δ. This basis gives us a simple and general formalization of the notion of constraint system as a pair (Σ, Δ).

Formal semantics of π-Calculus for SoS: labeled transition rules for behaviors

Restriction:

$$\frac{\text{constrainedBehavior}_1 \xrightarrow{\text{action}_1} \text{constrainedBehavior}_1{}'}{\textbf{value}\ \text{value}_1\ .\ \text{constrainedBehavior}_1 \xrightarrow{\text{action}_1} \textbf{value}\ \text{value}_1\ .\ \text{constrainedBehavior}_1{}'}$$

where value₁ $\notin$ *names*(action₁), *i.e.* value₁ *is not among the names used in* action₁

Communication:

$$\frac{\text{behavior}_1 \xrightarrow{\textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{value}_1} \text{behavior}_1{}' \qquad \text{behavior}_2 \xrightarrow{\textbf{via}\ \text{connection}_2\ \textbf{receive}\ \text{value}} \text{behavior}_2{}'}{\textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\ (\text{connection}_1 = \text{connection}_2)\\ \textbf{and}\ \text{behavior}_1\ \textbf{and}\ \text{behavior}_2\end{array}\right\} \xrightarrow{\tau} \textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\ (\text{connection}_1 = \text{connection}_2)\\ \textbf{and}\ (\text{value} = \text{value}_1)\ \textbf{and}\ \text{behavior}_1{}'\ \textbf{and}\ \text{behavior}_2{}'\end{array}\right\}}$$

where connection₁ = connection₂, *i.e.* (connection₁ = connection₂) *is a binding resulting from an extrusion or unification*

Restriction-Open:

$$\frac{\text{constrainedBehavior}_1 \xrightarrow{\textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{value}_1} \text{constrainedBehavior}_1{}'}{\textbf{value}\ \text{value}_1\ .\ \text{constrainedBehavior}_1 \xrightarrow{\textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{value}_1} \text{constrainedBehavior}_1{}'}$$

where value₁ ≠ connection₁, *i.e.* value₁ *cannot be used for connection as it is restricted*

Communication-Close:

$$\frac{\text{behavior}_1 \xrightarrow{\textbf{value}\ \text{connection}\ .\ \textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{connection}} \text{behavior}_1{}' \qquad \text{behavior}_2 \xrightarrow{\textbf{via}\ \text{connection}_2\ \textbf{receive}\ \text{value}} \text{behavior}_2{}'}{\textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\ (\text{connection}_1{=}\text{connection}_2)\\ \textbf{and}\ \text{behavior}_1\ \textbf{and}\ \text{behavior}_2\end{array}\right\} \xrightarrow{\tau} \textbf{value}\ \text{connection}\ .\ \textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\ (\text{connection}_1{=}\text{connection}_2)\\ \textbf{and}\ (\text{value} = \text{connection})\\ \textbf{and}\ \text{behavior}_1{}'\ \textbf{and}\ \text{behavior}_2{}'\end{array}\right\}}$$

where value $\notin$ *free*(behavior₂), *i.e.* value *is not restricted in* behavior₂ *while* connection *is restricted in* behavior₁

Choice:

$$\frac{\text{constraint}_{0..n}\ \textbf{and}\ (\text{action}_i\ .\ \text{behavior}_i{}') \xrightarrow{\text{action}_i} \text{constraint}_{0..n}{}'\ \textbf{and}\ \text{behavior}_i{}'}{\textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and choose}\left\{\text{action}_0\ .\ \text{behavior}_0{}' \ldots\ \textbf{or}\ \ \text{action}_m\ .\ \text{behavior}_m{}'\right\}\end{array}\right\} \xrightarrow{\text{action}_i} \textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}{}'\\ \textbf{and}\ \text{behavior}_i{}'\end{array}\right\}}$$

where *i* ∈ *0..m*, *i.e. only one of the actions* action₀..ₘ *is performed*

Conditional-Then:

$$\frac{\text{behavior}_1 \xrightarrow{\text{action}_1} \text{behavior}_1{}' \qquad \text{constraint} \equiv \text{true}}{\textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ (\textbf{if}\ \text{constraint}\ \textbf{then}\ \text{behavior}_1\ \textbf{else}\ \text{behavior}_2)\right\} \xrightarrow{\text{action}_1} \textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ \ \text{behavior}_1\right.}$$

Conditional-Else:

$$\frac{\text{behavior}_2 \xrightarrow{\text{action}_2} \text{behavior}_2{}' \qquad \text{constraint} \equiv \text{false}}{\textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ (\textbf{if}\ \text{constraint}\ \textbf{then}\ \text{behavior}_1\ \textbf{else}\ \text{behavior}_2)\right\} \xrightarrow{\text{action}_1} \textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ \text{behavior}_2{}'\right.}$$

Repetition:

$$\frac{\text{behavior}_1 \xrightarrow{\text{action}_1} \text{behavior}_1{}'}{\textbf{repeat}\left\{\text{behavior}_1\right\} \xrightarrow{\text{action}_1} \text{behavior}_1{}'\ .\ \textbf{repeat}\left\{\text{behavior}_1\right\}}$$

where behavior₁' . behavior₁ *is a sequential composition, i.e.* behavior₁' *must be performed before* behavior₁

Composition:

$$\frac{\text{constrainedBehavior}_i \xrightarrow{\text{action}_i} \text{constrainedBehavior}_i{}'}{\textbf{compose}\left\{\begin{array}{l}\text{constrainedBehavior}_0\ \textbf{...}\\ \textbf{and}\ \text{constrainedBehavior}_i\\ \textbf{and}\ \text{constrainedBehavior}_n\end{array}\right\} \xrightarrow{\text{action}_i} \textbf{compose}\left\{\begin{array}{l}\text{constrainedBehavior}_0\ \textbf{...}\\ \textbf{and}\ \text{constrainedBehavior}_i{}'\\ \textbf{and}\ \text{constrainedBehavior}_n\end{array}\right\}}$$

where *i* ∈ *1..n* **and** *bound*(actionᵢ) ∩ *free*(constrainedBehavior₀..ₙ₋ᵢ) = ∅,
i.e. restricted names in actionᵢ *are not restricted elsewhere*

Figure 5: Formal semantics of behaviors in architectural abstractions.

Let us take as basis the constraint system over finite domains (FD) [9]. In FD, variables are assumed to range over finite domains and, in addition to equality, we may have predicates that restrict the possible values of a variable to some finite set.

More formally, FD[n] (n > 0) is the constraint system where Σ is given by the constant symbols *0..n-1* as well as by the equality =, and Δ is given by the axioms of equational theory *x = x, x = y □ y = x, x = y □ y = z □ x = z , and v = w □ false* for each two different constants *v, w □* Σ.

In the π-Calculus for SoS, the constraint system is defined by the pair (Σ, Δ), where:

- Σ is defined by the set of constant symbols declared as values in the π-Calculus for SoS, the set of function symbols declared in the value types, and the set of predicate symbols equally declared in the value types, including built-in datatypes.
- Δ is defined by a first-order theory over Σ according to the value types defined in the π-Calculus for SoS using the logical operators shown in Figure 6.

The logical operators shown in Figure 6 include data (value) and channel binding predicates. Value predicates are expressed with relational operators on data expressions. Binding predicates describes the two possible channel bindings: *unify* (it is true if two connections are bound together by an equality constraint between internal connections of different constituents) and *relay* (it is true if two connections are bound together by an equality constraint between an external connection and an internal one of a constituent and the encompassing SoS.

Constraints on the channel bindings can also be expressed in terms of multiplicity. Universal and existential quantification are supported on all kinds of values.

2.4 Summing Up the Formal Definition of the π-Calculus for SoS

By its formal definition, the π-Calculus for SoS extends the original π-Calculus with constructs for handling concurrent constraints supporting mediation, where the mediation will be realized by inferred bindings from the set of constraints told to the environment.

Note that, differently from the foundations of single system ADLs which is constituted by process calculi such as FSP, CSP and π-Calculus, which declare the possible bindings between behaviors at design-time, the π-Calculus for SoS automatically generates them at run-time, by solving the told constraints (for details on our constraint solving mechanisms see [13]).

It is also worth noting that the formal operational semantics of the π-Calculus for SoS, defined by the labeled transition rules under concurrent constraints, enable both static and dynamic verification of properties.

In addition to structural properties derived by the topology of behaviors linked by inferred bindings, behavior properties related to safety and liveness can be verified, as well as absence of deadlock and livelock. Thus the π-Calculus for SoS allows these SoS architectural models to be verified semantically correct in terms of declared behaviors and properties.

Besides enabling verification, the formal semantics of the π-Calculus for SoS supports validation by symbolic execution as well as validation by executing appropriate use cases, including simulation. Moreover, communicating concurrent behaviors make possible to achieve emergent behaviors through the mediated interaction of composed behaviors.

Abstract syntax of constraints

predicate ::= valuePredicate | bindingPredicate	
valuePredicate ::= $\text{value}_1 = \text{value}_2$	*equality* $v_1 = v_2$
| value_1 <> value_2	*inequality* $v_1 \neq v_2$
| $\text{value}_1 < \text{value}_2$	*is less than* $v_1 < v_2$
| $\text{value}_1 > \text{value}_2$	*is greater than* $v_1 > v_2$
| value_1 <= value_2	*is less than or equal to* $v_1 \leq v_2$
| value_1 >= value_2	*is* greater *than or equal* $v_1 \geq v_2$
| **true** | **false**	*booleans true*, *false*
bindingPredicate ::=	
unify multiplicity_1 **{** connection_1 **} to** multiplicity_2 **{** connection_2 **}**	
| **relay** multiplicity_1 **{** connection_1 **} to** multiplicity_2 **{** connection_2 **}**	
multiplicity ::= **[one** | **none** | **lone** | **any** | **some** | **all]**	
constraint ::= predicate	
| **not** constraint_1	*negation* $\neg c_1$
| constraint_1 **and** constraint_2	*conjunction* $c_1 \wedge c_2$
| constraint_1 **or** constraint_2	*disjunction* $c_1 \vee c_2$
| constraint_1 **xor** constraint_2	*exclusive or* $c_1 \veebar c_2$
| constraint_1 **imply** constraint_2	*implication* $c_1 \Rightarrow c_2$
| **exists {** name_1 **in** sequence_1 **…,** name_n **in** sequence_n **suchthat** constraint_1 **}**	*existential quantification* $\exists x_1 \mid c_1$
| **forall {** name_1 **in** sequence_1 **…,** name_n **in** sequence_n **suchthat** constraint_1 **}**	*universal quantification* $\forall x_1 \mid c_1$
| valuing_1 . constraint_1	*restriction on constraint* $(x_1)\ c_1$

Figure 6: Abstract syntax of the constraint language.

3. Applying the π-Calculus for SoS in a Case Study

We will now present how the π-Calculus for SoS can be applied in practice for modeling SoS architectures through a case study drawn from an SoS for Flood Monitoring and Emergency Response.

3.1 Case Study: Flood Monitoring and Emergency Response SoS

Flood Monitoring and Emergency Response SoSs address the problem of flash floods, which constitute a significant threat in different countries during rainy seasons. This becomes particularly critical in cities that are crossed by rivers such as the city of Sao Carlos, SP, Brazil, crossed by the Monjolinho river as shown in Figure 7.

To address this major problem, we have architected with SosADL, based on the π-Calculus for SoS, a Flood Monitoring and Emergency Response SoS [8], including Wireless River Sensors, Telecommunication Gateways, Unmanned Aerial Vehicles (UAVs), Vehicular Ad Hoc Networks (VANETs), Meteorological Centers, Fire and Rescue Services, Hospitals, Police Departments, SMS Centers and Social Networks.

To highlight the main concepts and constructs of the π-Calculus for SoS, we will use a subset of this Flood Monitoring and Emergency Response SoS, which is itself an SoS, i.e. the Urban River Monitoring SoS. Indeed, an SoS may have constituent systems that are themselves SoSs.

The Urban River Monitoring SoS is based on two kinds of constituent systems: wireless river sensors (for measuring river level depth via physical pressure sensing) and a gateway base station (for analyzing variations of river level depths and warning on the risk of flash flood).

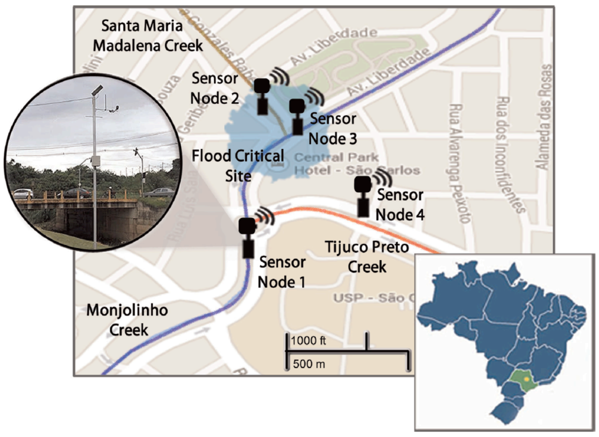


Figure 7: Monjolinho river crossing the city of Sao Carlos with deployed wireless river sensors.

A sensor node is an independent single system having as components a microcontroller, transceiver, external memory, power source and different sensory devices. Every sensor node periodically senses river pressure (which is then converted in river depth) and transmits this measure towards a gateway station, also an independent single system (note that the gateway station is in the range of transmission of some sensor nodes, but not of all of them). The gateway station examines the flow of measures in order to estimate if there is an imminent risk of flash flood.

Each sensor node acts as a peer to the other nodes and all together (as an SoS) has the capability of measuring the depth level of the river. The raw sensory data inputting from the environment (in this case, the river) first undergoes some processing in the sensor node and then is forwarded to neighboring sensor peers that retransmit the data.

In terms of the concrete implementation in the Monjolinho river, sensor nodes use the XBee technology and connections between nodes are materialized by ZigBee transmissions between sensors or between sensors and the gateway. These transmissions are ad-hoc and constrained by the distance between sensor neighbors and the distributed sensors and the gateway. They are also constrained by the remaining battery power of sensor nodes.

The gateway (an industrial computer linked to internet) provides the base station for collecting and processing measurements, possibly warning of the risk of imminent flood.

For achieving the stated mission by creating emergent behavior, the architecture of this Urban River Monitoring SoS needs to be rigorously designed. In particular, resilience (even in case of low charge in the battery of sensors) needs to be managed as well as its operation in an energy-efficient way.

This Urban River Monitoring SoS has the five defining characteristics of an SoS. Each sensor node operates in a way that is independent of other sensor nodes, as they belong to different city councils and have different missions in the metropolitan region of Sao Carlos, e.g. pollution control or water supply. Each one has its own management strategy for transmission vs. energy consumption and will act under the authority of the different city councils. New sensor nodes may be installed by the different councils as well as existing ones may be changed or uninstalled without any control from the SoS. Finally, the emergent behavior of flood detection is produced as the resulting behavior of multihop transmissions between distributed sensors and some of these sensors and the gateway that analyzes transmitted data.

The architectural model of an SoS is specified at two levels: the abstract architecture description (it defines the possible kinds of constituent systems that may participate in the SoS) and the concrete architecture (it instantiates the abstract architecture according to the selected constituent systems in a specific environment, in our case it defines the concrete SoS architecture for monitoring the Monjolinho river).

In term of the π-Calculus for SoS, it means that we will express the process definitions that specify the abstract architecture description and then apply these definitions in a specific context to obtain a concrete architecture description.

3.2 Modeling an abstract SoS architecture with the π-Calculus for SoS

Let us model the SoS abstract architecture focusing on the Urban River Monitoring SoS and then its concretization in the case of the Monjolinho river.

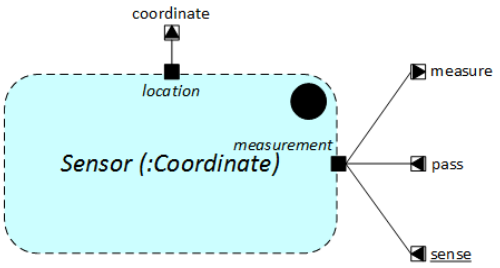
In the π-Calculus for SoS, the different kinds of constituent systems are declared as process definitions as shown in Figures 8, 9, and 10.

The *Sensor* process definition declares three gates: *measurement* that comprises connections for handling measures, *location* that include a connection for handling GPS coordinates, and *energy* that has connections for managing energy consumption. The behavior exposed by the *Sensor* system is shown in Figure 8, the diagram showing a subset of gates.

This behavior specification first declares a variable to store the value of the sensor's GPS coordinate, and then tell publicly this value sending it via its gate *location* to its neighbors. Subsequently, it receives the power threshold enabling operation and repeatedly (if the remaining power is greater than the power threshold) tells that it is able to operate and will choose between sensing raw data from the sensor device and transmitting the corresponding measure (sending a tuple with both its GPS coordinate and the measure converted from the raw data received) or just forward a measure received from a neighbor sensor via connection *pass*. If not, it does nothing, but continues to scan the power level of the battery to test if the remaining power became greater than the threshold. Whenever it is the case, it will be able to operate again.

Note that by this process definition every sensor node guarantees that, if operating, it will always be able to transmit measures (own measures or ones from its neighbors) via the connection *measure* of *measurement* gate. It is the case whenever the power available in the sensor node is greater than the power threshold, and if not, it will wait for the battery to sufficiently recharge again (in the case of the Monjolinho river, some sensor nodes include batteries charged by solar panels). Once recharged, it will continue to operate as guaranteed.

Note that by this process definition every sensor node guarantees that, if operating, it will always be able to transmit measures (own measures or ones from its neighbors) via the connection *measure* of *measurement* gate. It is the case whenever the power available in the sensor node is greater than the power threshold, and if not, it will wait for the battery to sufficiently recharge again (in the case of the Monjolinho river, some sensor nodes include batteries charged by solar panels). Once recharged, it will continue to operate as guaranteed.

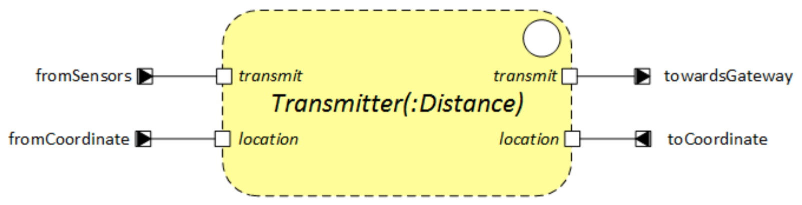


```
system Sensor(lps: Coordinate) is { …
  behavior sensing is {
     // 'behavior' executes when the 'system' is created
     value sensorcoordinate is Coordinate = lps
     tell sensorlocation is {sensorcoordinate = lps}
        // assert the specified constraint with name
        // and type extrusions
     via location::coordinate send sensorcoordinate
     via energy::threshold receive powerthreshold
     repeat {
        via energy::power receive powerlevel
        if (powerlevel > powerthreshold) then {
           // 'if' tests the condition without blocking
           tell powering is {powerlevel > powerthreshold}
           choose{
              via measurement::sense receive data
              via measurement::measure send
                 tuple{coordinate=lps,depth=data::convert()}
           } or {
              via measurement::pass receive data
              via measurement::measure send data
           }
        }
     }
  }
}
```

Figure 8: Excerpt of the *Sensor* system with behavior described in the π-Calculus for SoS.

Let us now define, in Figure 9, the *Transmitter* mediator, whose purpose is to transmit measures from each sensor towards the gateway using multihop transmissions. It has a duty to be fulfilled by gates of constituent systems. In the *Transmitter* mediator, the purpose is to bind on the one hand with a constituent system having a gate that will commit to fulfill the duty of providing measures to the mediator through connection *fromSensors* and on the other hand with another constituent system having a gate for consuming measures from the mediator through connection *towardsGateway*. The commitment of the mediator itself (see Figure 9) is to forward from the input connection to the output one without any processing in-between, if and only if the distance calculated from the GPS coordinates of these two bound sensors is in the range of transmission of the mediator.

The *Gateway* system is shown in Figure 10. For the sake of brevity, we will not detail its behavior.



```
mediator Transmitter(distancebetweengates:Distance) is { …
  behavior transmitting is {
     via location::fromCoordinate receive sendercoordinate
     via location::toCoordinate receive receivercoordinate
     ask sendercoordinate::distance(receivercoordinate)
        < distancebetweengates
     repeat {
           via transmit::fromSensors receive measure
           via transmit::towardsGateway send measure
     }
  }
}
```

Figure 9: Excerpt of the *Transmitter* mediator with behavior described in the π-Calculus for SoS.

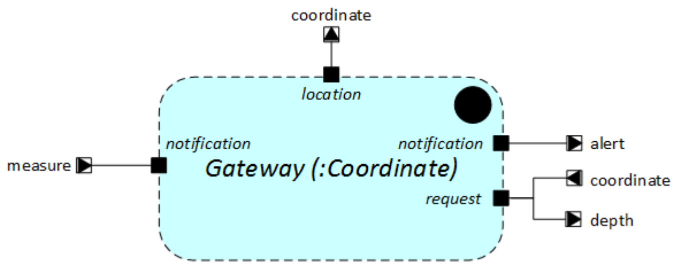


Figure 10: Excerpt of the *Gateway* system with behavior described in the π-Calculus for SoS.

Let us now describe the SoS abstract architecture, focusing on the declaration of the coalition of constituent systems related through mediators, by expressing the policies defining possible inferred channel bindings. As shown in Figure 11, a coalition forming this SoS architecture may involve possibly many sensor constituents, exactly one gateway constituent and possibly many transmitter mediators.

This coalition (expressed by a composite process definition with constraints, shown in Figure 11) does not specify which constituent systems will exist at run-time, but simply which are possible systems that may exist and which are the required conditions for forming a coalition among the systems identified at run-time to participate in the SoS. It does not specify either which concrete system will be attached to which concrete system through a mediator. It simply specifies what constraints must be satisfied.

The SoS architecture is thereby intentionally described. This suitably copes with the intrinsic SoS characteristics.

```
architecture WnsMonitoringSosArchitecture() is {…
  behavior coalition is compose {
     sensors is sequence{Sensor}
     gateway is Gateway
     transmitters is sequence{Transmitter}
  } binding {…
     forall{isensor1 in sensors, isensor2 in sensors
        suchthat
           exits{itransmitter in transmitters
              suchthat
                 (isensor1 <> isensor2) implies
                 unify one{itransmitter::fromSensors}
                     to one{isensor1::measurement::measure}
                 and unify one{itransmitter::towardsGateway}
                     to (one{isensor2::measurement::pass}
                 xor unify one{itransmitter::towardsGateway}
                     to one{gateway::notification::measure}
           }
     // multiplicities are 'one', 'none',
     // 'lone' (none or one),
     // 'any' (none or more),
     // 'some' (one or more), 'all'
     }
} guarantee {…}
```

Figure 11: Excerpt of an SoS architecture with binding constraints described in the π-Calculus for SoS.

3.3 Modeling a concrete SoS architecture with the π-Calculus for SoS

Let us now illustrate a typical scenario supported by the operational semantics of the π-Calculus for SoS. It is worth noting that it is the behavioral description of an SoS architecture which is the major differentiator when compared to architectures of single systems. Indeed, an SoS is specially characterized by emergent behaviors and it is by the analysis of emergent behaviors that SoS architectures will be purposely constructed and validated.

Let us take as an example the concrete architecture of the Urban River Monitoring SoS deployed in the operational environment of the Monjolinho river (shown in Figure 7). Based on the defined SoS abstract architecture with intentional channel bindings (see Figure 11), the operational environment of the Urban River Monitoring SoS is initially explored for identifying the actual sensors that will become constituents of this SoS (see [13] for the description of the mechanisms for synthesizing concrete SoS architectures based on process definitions in the π-Calculus for SoS). The concrete SoS architecture deployed in the Monjolinho river results from the selection at run-time of possible concrete systems that may participate in the SoS. In the case of our scenario to the Monjolinho river, shown in Figure 12, there are several sensors installed (of which 5 are selected as constituents of the SoS) and 1 gateway (also selected). Some sensors are too far from the gateway to transmit measures directly, needing to perform a multihop using intermediate sensors (it is the case of sensors 1, 2 and 3).

For constructing this initial concrete architecture, mediators were synthesized and channels bound taking into account the range of transmission of the created mediators and the geographical location (based on GPS coordinates) of the constituent systems. In this case, as shown in Figure 12, the concrete inferred bindings link, directly or transitively, all sensor nodes to the gateway base station.

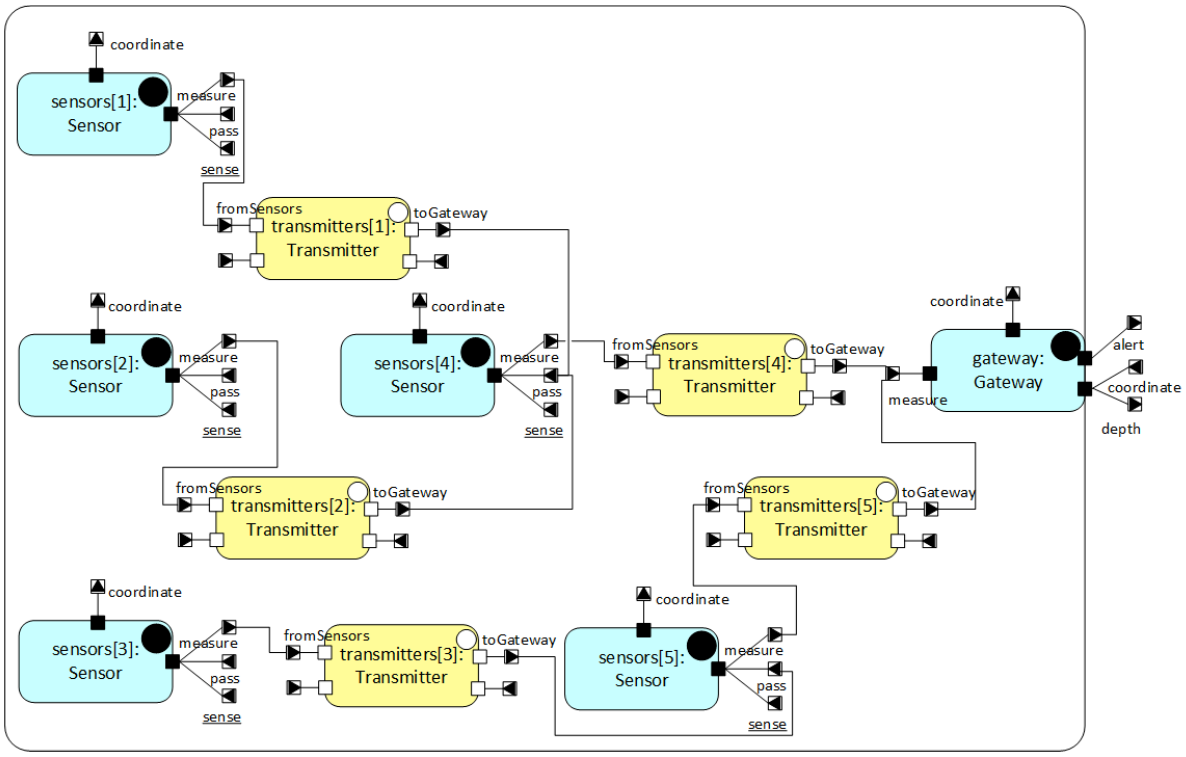


Figure 12: SoS concrete architecture: a coalition satisfying the constraints of the SoS abstract architecture.

Based on this concrete SoS architecture, the Urban River Monitoring SoS deployed in the Monjolinho river, is enabled to continuously achieve its mission by evolutionary development, automatically supporting the re-architecture of the SoS for satisfying the set of told constraints (see [13] for details on the constraint solving mechanism), sending an alert when the analysis performed by the gateway recognizes a risk of flash flood.

3.4 Evolutionary behavior of concrete SoS architectures with the π-Calculus for SoS

Continuing our illustrative scenario, let us now focus on the operational semantics of concurrent behaviors for understanding the emergent behavior of the SoS architecture formalized with the π-Calculus for SoS in the case of the Urban River Monitoring SoS deployed in the Monjolinho river.

We will concentrate first on the behavior and interactions of sensors. Let us emulate a sensor behavior (see Figure 8): after elaborating the value *sensorcoordinate* as the GPS position of the sensor (passed as parameter in its creation), the sensor tells to its local environment its geographical coordinates and then waits to send via the connection *location::coordinate* this value. The same specified behavior is executed in the five sensors, i.e. the five sensors are identified as constituent systems and started their execution. All five will reach their reduction limit and wait at the connection *location::coordinate*.

We will now emulate a transmitter behavior. Figure 9 shows that when a transmitter is created, it first waits to get the coordinate of the sensor that will send the data at *location::fromCoordinate* and, when received, waits the coordinate of the sensor that will receive the data at *location::toCoordinate*. When received, the mediator asks the environment whether the constraint “sender and receiver must be located in a distance less than the range of wireless transmission (value passed as a parameter of the mediator)” is inferred from the set of constraints told to the environment (gathering the different local environments). If it is the case, the transmitter mediator will, repeatedly, get as input the sensed value via its connection *transmit::fromSensors* and then output the sensed value via its connection *transmit::towardsGateway*.

Coming back to the sensor behavior (shown in Figure 8): when a mediator is created between two sensors (in our case, as shown in Figure 12, *transmitters*[1] is created between *sensors*[1] and *sensors*[4], *transmitters*[2] between *sensors*[2] and *sensors*[4], *transmitters*[3] between *sensors*[3] and *sensors*[5], *transmitters*[4] between *sensors*[4] and *gateway*, and *transmitters*[5] between *sensors*[5] and *gateway*), each one will get the coordinates of the source and destination sensors via its connection *location::fromCoordinate* and *location::toCoordinate* respectively, then ask if the connected sensors are in its transmission range. This is the case for all the sensors in this case, each mediator will repeatedly transmit the data from its source sensor to its destination one.

Let us now explain how the mediators were created using the information of the constituent systems and the role of mediators in the coalition. We will now emulate the coalition creation (see Figure 11): possible coalitions are declaratively expressed by a set of constraints on the possible compositions of constituent systems connected through mediators.

The declarative expression of possible coalitions shown in Figure 11 specifies that, for a number of identified sensors (arranged as a sequence) and a gateway, a concrete architectural coalition must, for each pair of constituent systems, have a transmitter mediator connecting sensors or (exclusive disjunction) connecting a sensor to the gateway.

Let us now come back to our example: the five sensors shown in Figure 12 are identified as constituent systems and their execution started. Based on these constituent systems, the SoS architecture solver implementing the π-Calculus for SoS will solve the Constraint Satisfaction Problem (CSP) (see again [13] for details): “which mediators need to be created to satisfy the set of finite constraints over variables specified in the coalition specification?”. Note that the π-Calculus for SoS supports dynamic CSPs as the set of constraints to consider evolves as well as decentralized CSPs as variables may be located at distinct constituents.

A solution to a CSP is an assignment of every variable by some value in its domain such that every constraint is satisfied. Therefore, each assignment of a value to a variable must be consistent (it must not violate any of the constraints). In the π-Calculus for SoS, the domain of variables are possible systems and possible mediators (knowing that the SoS can create mediators, but not systems). There can be multiple solutions (or none). The solution minimizing the number of mediators is currently preferred.

The solver takes the set of constraints of a coalition specification and finds architectural coalitions that satisfy them based on selected constituent systems. Coming back to our scenario, the solver created five mediators (shown in Figure 12) for connecting each of the sensors with another one (or the gateway) constituting paths that directly or transitively achieves the gateway.

It is important to highlight that the CSP mechanism used is incremental and scalable. In particular, it supports the exploration of alternative concrete architectures that could be selected according to utility functions in its initial configuration as well as incremental evolutions. For instance, if *sensors*[4] has a shortage of energy to continue to operate, the SoS will incrementally search for an incremental evolution of the initial configuration to continue to achieve emergent behavior (in this example having all sensors feeding the gateway). The evolved SoS architecture could be a reconfiguration where the transmitter between *sensors*[1] and *sensors*[4] is rerouted to *sensors*[2] and both *sensors*[4] and *transmitters*[4] are excluded.

Note thereby that the deployed concrete SoS architecture will evolve itself to continuously cope with the declared abstract SoS architecture.

4. Implementation and Validation of the π-Calculus for SoS

A major impetus behind developing formal foundations for SoS architecture description is that formality renders them suitable to be manipulated by software tools. The usefulness of providing the π-Calculus for SoS underlying SosADL is thereby directly related to the tools it provides to support architecture modeling, but also analysis and evolution, in particular in the case of SoSs. We have developed an SoS Architecture Development Environment for supporting architecture-centric formal development of SoSs based on the π-Calculus for SoS.

4.1 The SoS Architecture Development Environment

We have developed an SoS Architecture Development Environment (ADE) for supporting architecture-centric formal development of SoSs based on the π-Calculus for SoS. This toolset, called SosADE, is constructed as plugins in Eclipse Mars (http://eclipse.org/mars/). It provides a model-driven architecture development environment where:

- the meta-model of the π-Calculus for SoS is defined in EMF/Ecore (http://eclipse.org/modeling/emf/);
- the textual concrete syntax provided by SosADL is expressed in Xtext (http://eclipse.org/Xtext/);
- the graphical concrete syntax is developed in Sirius (http://eclipse.org/sirius/);
- the type checker is implemented in Xtend (http://www.eclipse.org/xtend/), after having being proved using the Coq proof assistant (http://coq.inria.fr/);
- transformations to input languages of different analysis tools supports validation and verification, including UPPAAL (http://www.uppaal.org/) for model checking, DEVS (http://www.ms4systems.com/) for simulation, and PLASMA (http://project.inria.fr/plasma-lab/) for statistical model checking [10][33].

The constraint solving mechanism implemented to support the *tell*, *ask*, *untell*, and *check* constraint handling constructs are based on the Kodkod SAT-solver (http://alloy.mit.edu/kodkod/). How these different tools were used for validating the π-Calculus for SoS as the formal foundation of SosADL is described in the next subsection.

4.2 Validating the π-Calculus for SoS in a Field Study (in vivo) of a real SoS

The π-Calculus for SoS, supported by its toolset, has been applied in different case studies and pilot projects for modeling SoS architectures. In particular, it has been applied for architecting the novel Flood Monitoring and Emergency Response SoS partially presented in this paper. Deployed in the Monjolinho river crossing the City of Sao Carlos, Brazil, it provided a real set for validating the π-Calculus for SoS as the formal foundation of SosADL and assessing its toolset.

(a) Description of the field (in vivo) study

Let us now briefly present the carried out field (in vivo) study of the Flood Monitoring and Emergency Response SoS in Table 1.

Field Study for Architecting a Flood Monitoring and Emergency Response SoS	
Purpose	The aim of this field study related to the development of a Flood Monitoring and Emergency Response SoS was to assess the fitness for purpose and the usefulness of the π-Calculus for SoS as formal foundation of SosADL to support the architectural design of real-scale SoSs.
Stakeholders	The SoS stakeholder is DAEE (Sao Paulo's Water and Electricity Department), a government organization of the State of Sao Paulo, responsible for managing water resources, including flood monitoring of urban rivers. This SoS also involves as stakeholders the different city councils crossed by the Monjolinho river, the policy and fire departments of the city of Sao Carlos that own the UAVs and have cars equipped with VANETs. Also involved are the hospitals of the city of Sao Carlos (they also have ambulances equipped with VANETs). The population, by downloading an App from the DAEE department, is also involved as target of the alert actions. They may also register for getting alert messages by SMS.
Mission	The mission of this SoS is to monitor potential floods and to handle related emergencies: detection of imminent floods and warning of citizens in risky areas.
Problem statement	In this field study, the wireless sensor network for urban river monitoring was enhanced with aerial and terrestrial vehicles: Unmanned Aerial Vehicles (UAVs) and Vehicular Ad-hoc Networks (VANETs). UAVs (microcopters with eight propellers and a camera) have as mission to enforce the resilience of the monitoring SoS, i.e. to maintain an acceptable level of service in the face of failures and problems to normal operation, by serving as router or serving as sensor data mule. They may also transmit images in real time for reducing the risks of false positives. In addition, the UAVs may provide data dissemination to VANETs embedded in vehicles crossing the risky area, thereby ensuring that vehicles driving towards a possible flood area can be warned to avoid certain roads.
Constituents	Constituent systems are: sensor nodes and a gateway in a Wireless Sensor Network (WSN); Unmanned Aerial Vehicles (UAVs); Vehicular Ad-hoc Networks (VANETs); SMS multicasting; DAEE Apps in Smartphones. Note that transmitter mediators are dynamically created to maintain the connectivity of the WSN.
Emergent behaviors	In order to fulfill its mission, this SoS needs to create and maintain an emergent behavior where sensor nodes (each including a sensor mote and an analog depth sensor) and microcopters (each including communication devices) will coordinate to enable an effective monitoring of the critical areas of the river and whenever a risk of flood is detected, to prepare the emergency response for vehicles approaching the flood area and inhabitants that live in potential flooding zones.
SoS architecture	The SoS architecture was described in SosADL (having the π-Calculus for SoS as its formal foundation) as a collaborative SoS: the architecture self-organize itself based on mediators for connecting sensors and forming multihop ad-hoc networks, using UAVs when needed, as well as always being ready to send flood alert messages towards VANETs or inhabitants.

Table 1. Field study of SosADL and the π-Calculus for SoS as its formal foundation.

(b) Reporting and lessons learned

As stated in Table 1, the aim of this field study was to assess the fitness for purpose and the usefulness of SosADL as textual notation and the π-Calculus for SoS as formal foundation to support the architectural modeling of real SoSs. The designed SoS abstract architecture for this case:

- was described in SosADL based on the π-Calculus for SoS;
- was edited using the SoS architecture *editor* (note that the excepts of the SoS architecture description in this paper are screen captures of the editor, depicted in Figure 13);
- was validated using the SoS architecture *validator* (by simulation in DEVS);
- had its assume-guarantee properties verified using the SoS architecture *verifier* (by model checking in UPPAAL);

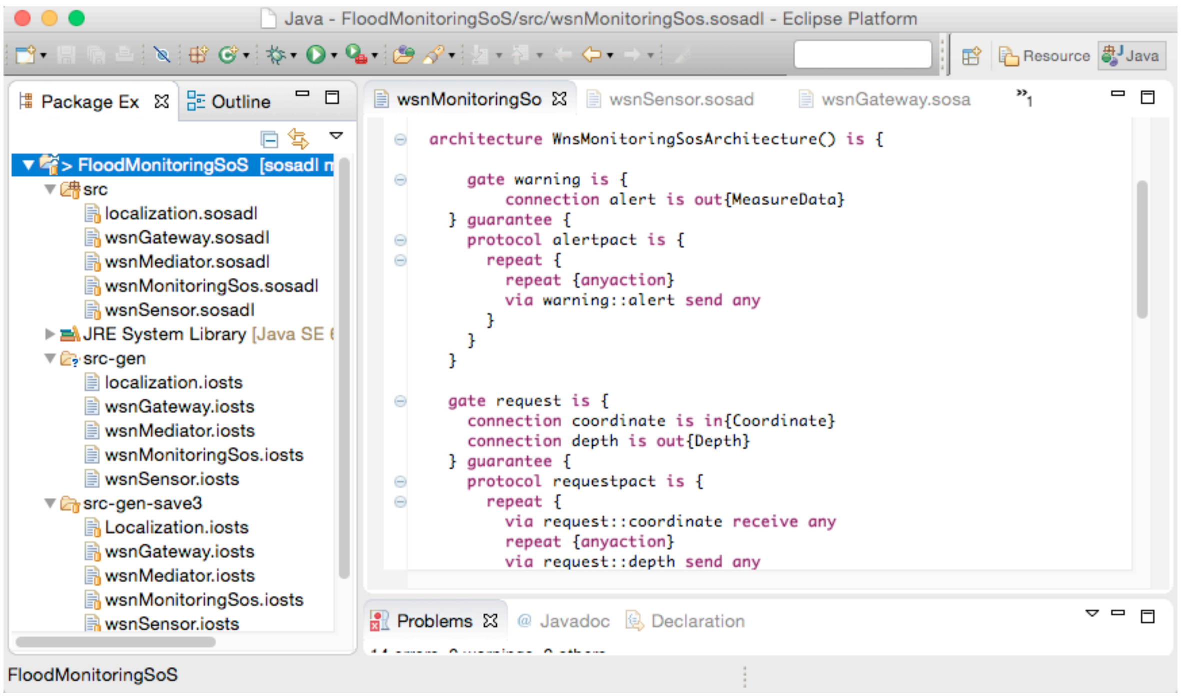
- had analysis of extreme cases due to the uncertainties of the river environment and actual availability of the constituent systems carried out using the SoS architecture *analyzer* (by statistical model checking in PLASMA).

The concretization of the SoS abstract architecture for the concrete case of the Monjolinho river was based on the actual installed sensors and gateway station. It was performed using the SoS architecture *constructor* (based on the Kodkod SAT-solver). Using the architecture-based *synthesizer*, we generated a concrete implementation of the SoS. Finally, the SoS architecture-based *evolver* (also based on the Kodkod SAT-solver) has been applied to assess the support for the evolutionary development of this SoS.

The result of this field study shows that the π-Calculus for SoS met the requirements for modeling SoS architectures and its emergent behaviors. As expected, using a formal ADL compels the SoS architects to study different architectural alternatives and take key architectural decisions based on SoS architecture analyses.

Learning the π-Calculus for SoS in its basic form was quite straightforward; however, using the advanced features of the process calculus needed interactions with the expert group. The SoS architecture *editor* and the *validator* were in practice the key tools in the learning and use of the π-Calculus for SoS and the *verifier* and *analyzer* were the key tools to show the added value of formally describing SoS architectures.

In fact, a key identified benefit of using the π-Calculus for SoS was the ability to validate and verify the studied SoS architectures very early in the application lifecycle with respect to the SoS correctness properties, in particular for studying the extreme conditions in which emergent behaviors were not able to satisfy the SoS mission.



```
architecture WnsMonitoringSosArchitecture() is {

    gate warning is {
        connection alert is out{MeasureData}
  } guarantee {
    protocol alertpact is {
      repeat {
        repeat {anyaction}
        via warning::alert send any
      }
    }
  }

  gate request is {
    connection coordinate is in{Coordinate}
    connection depth is out{Depth}
  } guarantee {
    protocol requestpact is {
      repeat {
        via request::coordinate receive any
        repeat {anyaction}
        via request::depth send any
```

Figure 13: SoS Architecture Development Environment (*editor*).

5. Conclusion and Future Work

This paper presented the π-Calculus for SoS, a π-Calculus with concurrent constraints and inferred channel bindings, specially designed for describing SoS architectures. It focused first on the needs of process calculi for modeling SoS architectures as formal foundation of ADLs, second on the approach for coping with these needs and third on the formal definition of the π-Calculus for SoS.

The significant difference between process calculi for single systems (i.e. FSP/Darwin, CSP/Wright, and π-Calculus/π-ADL) and the π-Calculus for SoS is in its treatment of exogenous, intentional, constrained and mediated channel bindings subject to uncertainty while enabling achievement of emergent behavior by the evolutionary interplay of the structure and behavior of concurrent systems in SoSs.

Its main contribution beyond the state-of-the-art in SoS is thereby to be the first process calculus having the expressiveness to address the challenge of modeling architectures of software-intensive SoSs from the architectural perspective. By its formal semantics, the π-Calculus for SoS supports automated verification of correctness properties of SoS architectures and supports validation through executable specifications.

The π-Calculus for SoS provides the formal foundation of a novel ADL for SoS, i.e. SosADL [26], exhibiting an architect-friendly notation, which has been applied in several case studies and pilots where the suitability of the formal system, the language and the supporting toolset have been validated.

Ongoing and future work is mainly related to the application of the π-Calculus for SoS based on its SosADL notation in industrial-scale projects. These include joint work applying the π-Calculus for SoS to the architectural modeling of naval SoSs in collaboration with DCNS, to the architectural modeling of SoS smart-farms in cooperative settings with IBM, and to architectural modeling in the transport domain with SEGULA.

Acknowledgement

The author would like to thank the four anonymous reviewers (in particular the second and fourth ones) for their careful reading, insightful comments and constructive suggestions that greatly helped to improve this paper.

References

[1] Allen, R.; Garlan, D.: "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6 (3), Jul. 1997, pp. 213-249.

[2] Aceto, L.; Fokkink, W. (Eds.): "Structural Operational Semantics", *The Journal of Logic & Algebraic Programming*, Jul.-Dec. 2004, pp. 1-464.

[3] Arnold, A; Boyer, B; Legay, A.: "Contracts and Behavioral Patterns for SoS: The EU IP DANSE approach", *Proc. of the ETAPS Workshop on Advances in Systems-of-Systems (AiSoS)*, Mar. 2013, pp. 47-66.

[4] Bengtson, J. et al.: "Psi-Calculi: Mobile Processes, Nominal Data, and Logic", *Proc. of the 24th IEEE Symposium on Logic in Computer Science (LICS)*, Aug. 2009, pp. 39-48.

[5] Buscemi, M.G.; Montanari, U.: "CC-Pi: A Constraint-based Language for Specifying Service Level Agreements", *Proc. of the 16th European Conference on Programming (ESOP)*, Mar. 2007, pp. 18-32.

[6] Cavalcante, E.; Batista, T.V.; Oquendo, F.: "Supporting Dynamic Software Architectures in π-ADL: From Architectural Description to Implementation", *Proc. of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, May 2015, pp. 31-40.

[7] Cavalcante, E.; Quilbeuf, J.; Traonouez, L.M.; Oquendo, F.; Batista, T.V.; Legay, A.: "Statistical Model Checking of Dynamic Software Architectures", *Proc. of the 10th European Conference on Software Architecture (ECSA)*, LNCS, Springer, September 2016.

[8] Degrossi, L.C. et al.: "Using Wireless Sensor Networks in the Sensor Web for Flood Monitoring in Brazil: Lessons Learned", *Proc. of the 10th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, May 2013, pp. 1-5.

[9] Fernandez, A.J.; Hill, P.M.: "A Comparative Study of Eight Constraint Programming Languages over the Boolean and Finite Domains", *International Journal on Constraints*, Vol. 5, No. 3, July 2000, pp. 275-301.

[10] Gabbrielli, M.; Palamidessi, C.; Valencia, F.D.: "Concurrent and Reactive Constraint Programming", *25-Year Perspective on Logic Programming*, LNCS 6125, Springer, 2010, pp. 231-253.

[11] Guessi, M. et al.: "Characterizing Architecture Description Languages for Software-Intensive Systems-of-Systems", *Proc. of the ACM/IEEE 3rd ICSE International Workshop on Software Engineering for Systems-of-Systems (SESoS)*, Eds. F. Oquendo et al., May 2015, pp. 1-8.

[12] Guessi, M.; Nakagawa, E.Y.; Oquendo, F.: "A Systematic Literature Review on the Description of Software Architectures for Systems-of-Systems", *Proc. of the 30th ACM Symposium on Applied Computing (SAC)*, Apr. 2015, pp. 1-8.

[13] Guessi, M.; Oquendo, F.; Nakagawa, E.Y.: "Checking the Architectural Feasibility of Systems-of-Systems using Formal Descriptions", *Proc. of the 11th System-of-Systems Engineering Conference (SoSE)*, June 2016.

[14] Hoare, C. A. R.: *Communicating Sequential Processes*, Prentice Hall, 2004, 260 p.

[15] ISO/IEC/IEEE 42010:2011: *Systems and Software Engineering – Architecture Description*, ISO, December 2011, 46 p.

[16] John, M. et al.: "The Attributed π-Calculus". *Computational Methods in Systems*, Springer, 2008, pp. 83-102.

[17] Kramer, J.; Magee, J.; Uchitel, S.: "Software Architecture Modeling and Analysis: A Rigorous Approach", *Formal Methods for Software Architectures*, Springer, 2003, pp. 44-51.

[18] Klein, J.; van Vliet, H.: "A Systematic Review of System-of-Systems Architecture Research", *Proc. of the 9th International Conference on Quality of Software architectures (QoSA)*, Jun. 2013, pp. 13-22.

[19] Maier, M. W.: "Architecting Principles for Systems-of-Systems", *Systems Engineering*, 1 (4), 1998, pp. 267-284.

[20] Magee, J.; Kramer, J.: *Concurrency: State Models and Java Programs*, Wiley, 2006, 434 p.

[21] Malavolta I. et al.: "What Industry Needs from Architectural Languages: A Survey", *IEEE Transactions on Software Engineering*, 39 (6), Jun. 2013, pp. 869-891.

[22] Medvidovic N.; Taylor R.: "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, 26 (1), Jan. 2000, pp. 70-93.

[23] Milner, R.: *Communicating and Mobile Systems: The π-Pi-Calculus*, Cambridge University Press, 1999.

[24] Olarte, C.; Rueda, C.; Valencia, F.D.: "Models and Emerging Trends of Concurrent Constraint Programming", *International Journal on Constraints*, 18 (4), Oct. 2013, pp. 535-578.

[25] Oquendo, F.: "π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures", *ACM Software Engineering*, 29 (3), May 2004.

[26] Oquendo, F.: "Formally Describing the Software Architecture of Systems-of-Systems with SosADL", *Proc. of the 11th System-of-Systems Engineering Conference (SoSE)*, June 2016.

[27] Oquendo, F.: "π-Calculus for SoS: A Foundation for Formally Describing Software-intensive Systems-of-Systems", *Proc. of the 11th IEEE System-of-Systems Engineering Conference (SoSE)*, June 2016.

[28] Oquendo, F.: "Software Architecture Challenges and Emerging Research in Software-intensive Systems-of-Systems", *Proc. of the 10th European Conference on Software Architecture (ECSA)*, LNCS, Springer, September 2016.

[29] Oquendo, F.: "Case Study on Formally Describing the Architecture of a Software-intensive System-of-Systems with SosADL", *Proc. of 15th IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, October 2016.

[30] Ozkaya M.; Kloukinas C.: "Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability", *Proc. of the 39th Euromicro Conference on Software Engineering & Advanced Applications (SEAA)*, Sept. 2013, pp. 177-184.

[31] Parrow, J.; Victor, B.: "The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes". *Proc. of the 13th IEEE Symposium on Logic in Computer Science (LICS)*, June 1998, pp. 176-185.

[32] Plotkin, G.: "A Calculus of Chemical Systems". *In Search of Elegance in the Theory and Practice of Computation*, Springer, 2013, pp.445-465.

[33] Quilbeuf, J.; Cavalcante, E.; Traonouez, L.M.; Oquendo, F.; Batista, T.V.; Legay, A.: "A Logic for Statistical Model Checking of Dynamic Software Architectures", *Proc. of the 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*,

Springer, October 2016.

[34] Regev, A.; Shapiro, E.: "The π-Calculus as an Abstraction for Biomolecular Systems". *Modelling in Molecular Biology*, Springer, 2004, pp. 219-266.

[35] Ritson, C.G.; Welch, P.H.: "A Process-oriented Architecture for Complex System Modelling", *Concurrency and Computation: Practice and Experience*, 2010, Vol. 22, pp. 965-980.

[36] Silva E.; Batista, T.; Oquendo, F.: "A Mission-Oriented Approach for Designing System-of-Systems", *Proc. of the 10th International Conference on System-of-Systems Engineering*, May 2015, pp. 346-351.

[37] Stirling, C.: *Modal and Temporal Properties of Processes*. Springer, 2001.

[38] Welch, P.H.; Barnes, F.R.M.: "Communicating Mobile Processes", *Communicating Sequential Processes: The First 25 Years*, LNCS 3525, Springer, 2005, pp. 175-210.

[39] Wischik, L.; Gardner, Ph.: "Explicit Fusions", *Theoretical Computer Science*, 340 (3), Aug. 2005, pp. 606-630.

[40] Woodcock, J. et al.: "Features of CML: A Formal Modelling Language for Systems-of-Systems", *Proc. of the 7th International Conference on Systems-of-Systems Engineering (SoSE)*, Jul. 2012, pp. 1-6.

[41] Woodcock, J. et al: "The COMPASS Modelling Language: Timed Semantics in UTP", *Communicating Process Architectures (CPA)*, Open Channel Publishing, 2014.