Available online at www.sciencedirect.com

ScienceDirect

Procedia Manufacturing 00 (2018) 000–000

Procedia MANUFACTURING

www.elsevier.com/locate/procedia

4th International Conference on System-Integrated Intelligence

A Unified System Modelling and Programming Language based on JavaScript and a Semantic Type System

Stefan Bosse

University of Bremen, Department of Mathematics & Computer Science,
University of Koblenz-Landau, Fac. Computer Science, Koblenz, Germany

Abstract The design and simulation of complex mechatronic and intelligent systems require a unified system modelling and programming language. This work introduces JavaScript as a unified modelling and programming language by extending JavaScript with a semantic type system extension JST as a possible solution to fill the gap between models and implementations, finally resulting in the JS+ super set language combining typing, modelling, and programming. The paper shows various model domains and their relation to the JS+ programming model including some generic transformation rules. Finally, a system compiler framework is introduced that can process JS+ models and program code. The tool uses JS+ input to produce a wide range of output formats for software and hardware design, and multi-domain simulation.

Keywords: System Modelling, Sensor Network, Agents, Embedded System Design, Modelling Language, Programming Language

1. Introduction

The design and modelling of complex and heterogeneous distributed sensor and actuator systems on different levels addressing different domains is a challenge. The central question. is how to model, design, simulate, and implement (program) complex Cyber-Physical Systems with one unified approach and language?

Examples for such systems are distributed Structural Health Monitoring (SHM) and Cyber Physical Systems (CPS) in the context of industrial production and manufacturing environments. :

Modelling is performed on different abstraction and functional levels:

- System-of-system;
- System;
- Embedded system;
- Operating system;
- Networking & Communication;
- Distributed Computing (e.g. using agent-based systems);
- Sensor and electronics;
- Physical,
- Hardware; and
- Software.

The different levels using commonly different modelling and programming languages. The difference between modelling and programming languages are summarizted in Fig. **1**.

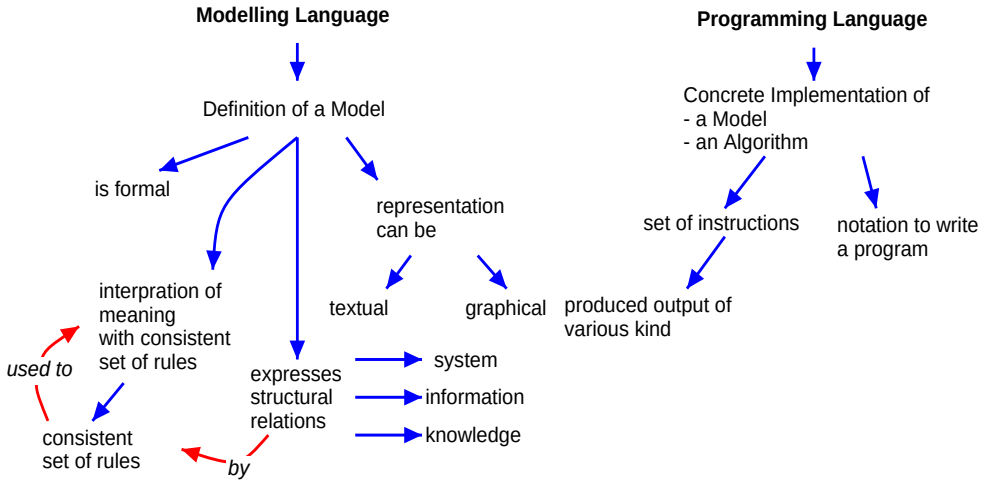

Fig. 1. Difference between Modelling and Programming Languages

A modelling expresses structure and relation by a consistent set of rules used for the interpretation of meaning. A programming lamguage is a concrete implementation of a model or algorithm by a set of instructions producing output of various kind.

There are already multiple modelling (M) and programming (implementation, I) languages and tools available:

- VHDL → Hardware Behaviour Modelling Languages for digital hardware (M,I)
- VHDL-AMS → Hardware Behaviour Modelling Language for digital and analog electronics harwdare (M,I)
- SystemC → Hardware and Software Modelling+Programming Language for digital hardware \u2192 algorithmic level addressing embedded systems (M,I)
- SystemC-AMS → Hardware and Software Modelling+Programming Language for digital and analoge systems
- JAVA → Object-orientated software programming only (I)
- C++ → Procedural and Object-orientated software programming only (I)
- SysML → Modelling (M) of software, hardware, and embedded systems
- Modelica → Object-oriented, declarative, multi-domain modeling language for component-oriented modeling of complex systems

Modelling and programming languages can pose a graphical or textual representation. Common textual representatons are *SystemC* **[1]** that is used for hardware-software co-design, *VHDL* (and *Verilog*) are used for modelling and engineering of hardware components (Digital logic and SoC designs, e.g., implementing sensor nodes, communication devices, and so on); *C*/*C*++/JAVA are used for software programming only; And Modelica, Simulink, and Matlab are used extensively for simulation and modelling of physical, electrical, numerical, and computational systems. Existing System Modelling Languages, e.g., *SysML*, including software-related *UML*, can be used to address different levels and layers in the design process of complex systems. But *SysML* is actually not a textual language and cannot be used for behavioural implementation, limited to specification and structural description only.

System-C is one of the promising unified modelling and programming languages to design complex embedded systems. In **[2]** an extended *SystemC* model (*SystemC*-A) was used to model a complex automotive system on multi-domain level. SystemC-A provides analogue, mixed-signal and mixed-domain modeling capabilities based on the *VHDL-AMS* extension. But even *SystemC*-A lacks specifications and model implementations on all domains as outlined in Sec. 2., e.g., specifications of electrical or mechanical simulations. Commonly there is a relevant gap between existing model capabilities (the model specifications) and concrete implementations.

VHDL-AMS was used in multi-domain simulation **[3]** and the design of mechatronics designs **[4]** on a suitable abstract level. But only electrical, physical, and logic components and their interaction can be modelled.

The design of material-integrated intelligent systems, e.g., smart adaptive materials **[5]**, addresses all possible domains, abstraction levels, and is characterized by complex interactions.

In this work *JS+* as an extension of the widely used and established programming language JavaScript (*JS*) is introduced serving as a universal modelling and programming language for the unique design of complex distributed systems addressing all levels and aspects of the system specification and implementation. The major advantage of *JS* over, e.g., *SystemC*, is the feature that *JS* can be used for the modelling and the implementation of synthesis tools, too. For example, the widely used *JS* parser *esprima* is written in *JS* and can be easily extended and customized. Using *JS* enables rapid development of tools as outlined in Sec. 4. *SystemC* is still a C++ library enabling designers to both implement and simulate a system using the library's structures and any C++ construct (that is supported by the compiler being used **[1]**.

In contrast, *JS* has no strong bindings to compilers or execution platforms (*JS* engines). *JS* do not require a program or module frame (i.e., like the main function and library environment in *C++*/*SystemC*). Any *JS* code snippet can be executed standalone. Data and code can be uniquely handled and exchanged by the *JSON+* file format (*JSON* extended with function code). And most relevant *JS* relies on three fundamental programming classes: Functional; Object-orientated; Procedural. Additionally, *JS* is a dynamic typed language and a program context and objects can be extended at run-time preventing an over-specification on model level. The advantage of using one widely used language instead of multiple different expert-level languages is that system modelling and implementation can be performed by a larger community of non-experts and application engineers. This is conducted by embedding the proposed approach in a toolbox framework enabling the unique design of complex distributed sensing and cyber-physical systems. Furthermore, this language unification simplifies the deployment of high-level synthesis approaches. *JS* is well suited for this approach due to its generic and portable programming model that can be executed on any device.

JS in its current form cannot be used directly for this purpose. To match *JS* with the requirements of a system modelling language, a Semantic Type System (*JST*) is introduced extending *JS* with formal specification capabilities. It can be shown that *JST* enables the transformation of common *JS* language constructs (objects, arrays, functions, variables) to different modelling types (package, generic component, structure definition, hardware component, program, computational process, agent behaviour, numerical function, parametric constraints, requirement, communication protocol, interaction, and many more). Finally, *JST* converts the originally dynamic typed programming language *JS* to a type-constrained programming language as a prerequisite for compliant programming satisfying specified model properties at run-time avoiding ambiguities. Semantic aspects of programming like sum types not supported by *JS* (and most modern programming languages) are added by *JST* and finally creating a super set *JS+* of *JS*, shown in Fig. .**2**.

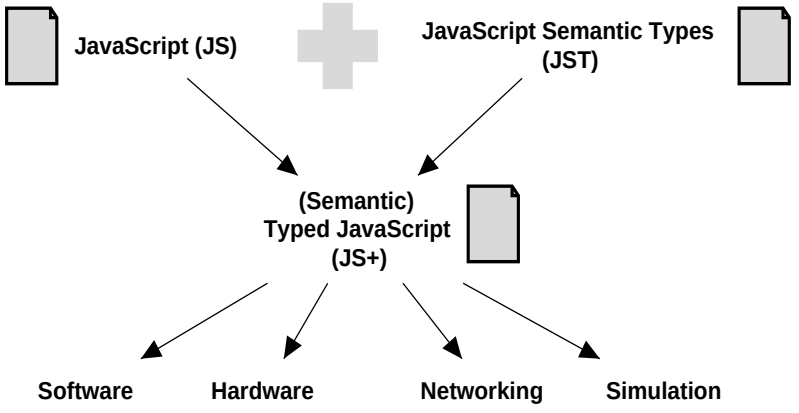

Fig. 2. Conceptual overview: One unified modelling and programming approach Software-Hardware-Complete!

Selected examples demonstrate the suitability of *JS*/*JST*/*JS+* for unified modelling of hardware components (digital logic and SoC designs), parallel software programming with a multi-process model and guarded resources, agent behaviour modelling and programming, system modeling of a sensor node and sensor networks, and mixed-domain simulation coupling computational and physical models, e.g., used for the design and evaluation of smart adaptive materials.

This paper is on one hand a proposal or draft of a unified multi-domain programming and modelling language showing selected highlights, on the other hand it poses existing tools that can already handle parts of *JST* and its programming and modelling system. Various examples are shown in App. A.

2. The Hierachical System Model

The entire system model that is represented by the semantic type system and that is used to design complex systems consists of multiple different domains:

1. Physical Level (Sensors, Mechanics, ..)
2. Hardware level (SoC, Analog & Digital Electronics, Sensors, ..)
3. Software level
4. Interaction level (Communication & Protocols)
5. System level
6. Simulation level

Each domain consists of multiple elements (components) representing types in the semantic type system. The system model can be extended. The entire system model that is represented by the semantic type system and that is used to design complex systems is shown in Fig. **3**. Fig. **3**. shows only some examples of domains and elements.

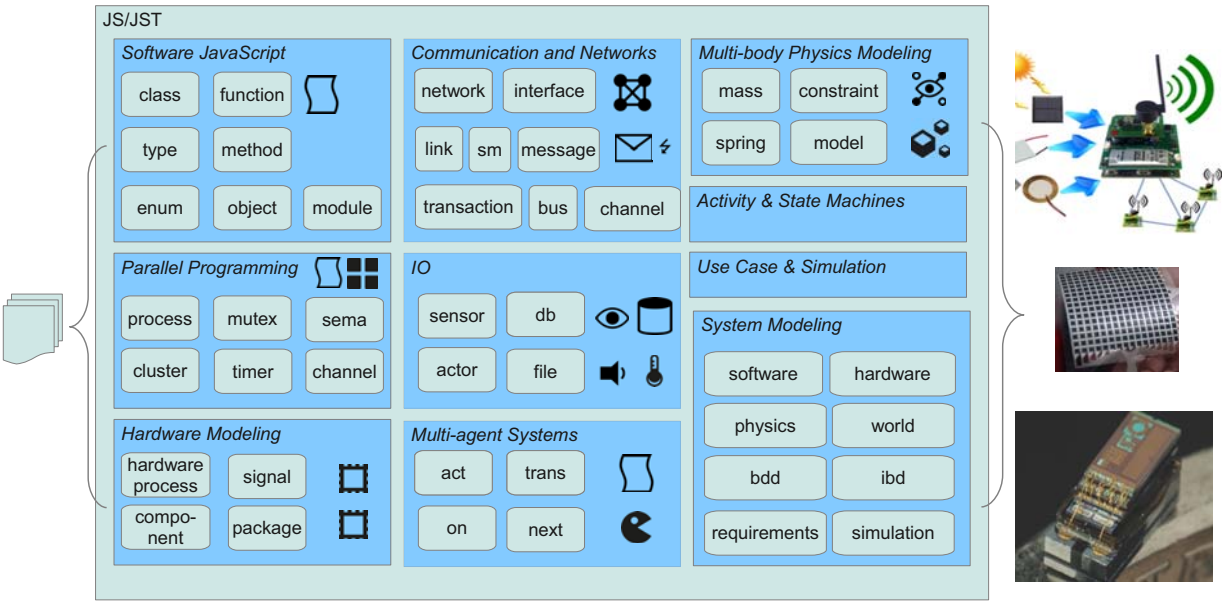

Fig. 3. The unified JS/JST system model as a toolkit for modeling complex systems composed of hardware & software modules. The domain boxes representing system modules and show some of their elements (model components).

3. The Semantic Type System

The basic idea behind the approach supplying a unified programming and modelling language for complex hardware-software systems (like sensor networks) is the (re)usage of core JavaScript *JS* with an extended type system *JST* providing semantic relations. The core concepts of the JavaScript programming model are generic functions and generic objects.

Association and definition of semantics can be done by:

7. Implicit Structural name-semantic convention, e.g. object attributes having a semantic meaning:

```
var simulationmodel = {
  physics : { ..},
  agents: { explorer:{..},..},
  parameter:{},  ..
}
```

8. Explicit type definitions, type extensions, type application, and type hierarchy (type sets):

```
type S = { .. };
class C = { .. }; node class N = {..};
network object = { n1:node class N = { .. }, ..};
process constructor function () {..} → type ..
sensor class straingauge = { .. }
communication protocol class = { .. }
```

Type extensions add semantic extensions to core type elements like functions, objects, and classes. For example, the process constructor function extensions associated a process model to the constructor function. The extended language system has two levels:

1. Standalone approach with *JS* and *JST*
2. *JST* integrated in *JS* creating an extended *JS+*.

On the first level, all programming and modelling components are specified entirely on programming level in pure JavaScript with separate and overlayed type declarations (type interfaces and type signatures) specifying the mapping of *JS* definitions on specific system model domains and component types (the semantic mapping, see Fig. **3**). The second level enables the integration of semantic types in *JS* directly and requires a distinct source code parser and analyzer (see Fig. **4** in Sec. 4.1.). Both levels are compared in the following example. Both representations can be transformed and are isomorphic.

JS

```
var x;
function p(a,b,c) {..}
function n(){ this.e_i=ε_0,..}
n.prototype.m_j=function(){..}
⇔
JS+
var x:process
process constructor function p(a,b,c) {..}
node class n = { e_i:t_i,m_j:method,..}
```

JST

```
x: process;
p: process constructor (a:typ_1,..)
n: node class n = { e_i:t_i,m_j:method,..}
```

Type in JST can be general (function, object, constructor, class) or related with a semantic meaning (process, hardware component, communication port, and so on). Associations can be orderd by the notation `type X => Y => Z`.with *X* as a superclass with strongest semantic binding of *Y*, and *Y* a superclass of *Z* (the most generic association).

In the following sub-sections different domains of the system model are presented with its *JS+*/*JST* representation and some examples of transformations performing a mapping of *JS+* to the respective domain specific target programming or modelling language *L*. In this context type declarations have to be performed by using a mapping function τ: *JST* → *L*. Basically there are three fundamental *JST* definitions and declarations: Types (`type`), Classes

(`class`), and Object (`object`). Classes and objects are extended by prefixes with respect to their domain and the domain elements they represent, e.g., `process class`, `network class`, `hardware component class`. Classes can be parameterized defining an object constructor function, i.e., `class` C`(`p`,..)`.

3.1. Software Programming Types

The first part of the *JS* Semantic Type System (*JST*) addresses the software programming level. The original intention of the *JST* was to extend *JS* with type annotations and to support abstract data types like sum types. It provides an extension to the *JS* core type system, which only consists of a few basic types, functions, and objects. The *JST* notation cannot be processed by the *JS* run-time system directly and requires an extended parser (*estprima*). *JST* is basically specified in a separate type interface file (with jsi extension) or can be embedded in *JS* comments for full compatibility. With an extended *estprima* parser that understands *JST* extensions it is possible to combine *JST* and *JS* code and compile native *JS* code. The original purpose was documentation, but the *JST* can be used for type checking and profiling using external tools or *JS* extensions. Initially, there is no type signature for an *JS* object or function. Although there is *JavaDoc* it is limited to the *JS* core type system. Commonly, a programming language covers only types that are immediately related to the language and run-time concepts, e.g., structures. But often there is a higher abstract type level composed of core types (abstract data types). One example is a sum type, supported only by a few languages directly. A sum type is composed of different types, i.e., structures or objects, that can be distinguished by pattern matching or type tags using names. Usually the sum type elements (the sub-types) are associated with a constructor function.

There are type definitions `type t = T` (equality) and type declarations of functions, objects, parameters, variables, `o : T` (assignment), which is equivalent to `typeof o = T`. Object types (with associated method prototypes) are denoted by the `object` keyword, whereas usual procedural records (without associated methods) are denoted by the `{}` notation. Arrays are either denoted with the `array` keyword or with the short notation `[]`.

Variable and Functions Type Constraints

Using *JST* variables, function parameters, and function results can be type constrained by adding type declaration statements or by adding type constraints in definition statements shown below. Declaration, definition, and implementation can be mixed.

Type Constraints
`var` *v*`:`*type*`;`
`function` *f*`(`p_1`:`typ_1`,`p_2`:`typ_2`,..):`*Return Type* `{` *statements* `}`

Type Signatures
v`:`*type*`;` `function` *f*`(`p_1`:`typ_1`,`p_2`:`typ_2`,..)` → *Return Type*

Typed Objects

In *JS+* an object can be defined with an embedded type signature combining implementation and type declaration:

```
var obj-name = {
  e1: typ1 = val1,
  e2: typ2 = val2,
  e3 = function f(p1:typ1,p2:typ2,..) {..} → typret
  ..
  en: typn = valn,
}
```

Sum Types

Most common programming languages support product types (structures, arrays), but lack of sum types - although used frequently - that are usually implemented with hooks using core types of the programming language. A sum type S is a disjunction type (in contrast to conjunction product types), defining a set of sub-types of objects $\{S_1,S_2,..\}$. *JS* does not support sum types, hence it is an abstract data type implemented with objects that require a specific tag attribute in the object sub-type identifying the sub-type uniquely. Generally, a generic sum-type declaration can be used in any other type declaration by using the alternation | character. For each sum type there is a constructor object *S*, and each sub-type S_x of *S* has a constructor function $S_x(e_1,e_2,..)$ (per convention a constructor function name starts

with an upper case letter). In *JS*, a sum type is associated with an object with attributes related to the sub-type tag (per convention a tag name starts with a lower case letter). The transformation rule *JS*+ →*L*(JS) is shown in Eq. **1**. The tag specification of each sub-type element is optional. If omitted a tag attribute with a string specifying the sum type and sub-type is added implicitly (e.g., "`S.S1`"). The synthesized *JS* sum type consists of the tag attributes and the constructor functions (see example in App. A.). Sum type definition templates are shown below.

Type Definition
`type` *S* `=`
  S_1 {*tag*=*S*.s_1,e_1:typ_1,e_2:typ_2,..} |
  S_2 {*tag*=*S*.s_2,e_1:typ_1,e_2:typ_2,..} |
  ..

Sub-type Constructor Function
S_1 {*tag*=*S*.s_1,e_1:typ_1,e_2:typ_2,..} *is a short form and synonym for*

`function` S_1(e_1:typ_1,e_2:typ_2,..) → *S*.S_1 {} ∈ *S*

⇔

Enumeration Object and Type Definition
`enum` *St* `= {`
  S_1 {*tag*=*S*.s_1,e_1:typ_1,e_2:typ_2,..},
  S_2 {*tag*=*S*.s_2,e_1:typ_1,e_2:typ_2,..},
  ..
} : *S*

`typeof` *St* = *S* ↔ `object` *St* : *S*

$$\frac{\text{type } S = S_i \ \{e_{i,1}{:}typ_{i,1},..\} \mid S_j \ \{e_{j,1}{:}typ_{j,1},..\} \mid ..}{\begin{array}{l}\text{var } S = \{\text{s}_i : \text{'}S.si\text{'},\text{s}_j : \text{'}S.sj\text{'},..,\\ S_i{:}\text{function}(p_{i,1},..)\{\text{return } \{\text{tag=S.s}_i,e_{i,1}{:}p_{i,1},..\}\},\\ S_j{:}\text{function}(p_{j,1},..)\{\text{return } \{\text{tag=S.s}_j,e_{j,1}{:}p_{j,1},..\}\},..\\ \}\end{array}} \quad (1)$$

Enumeration Types

An enumeration type is a simple symbolic sum type. In *JS* enumeration is performed usually by defining an object consisting of constant value properties. In *JST*, only the enumeration symbol names must be specified automatically assigning concrete values (optionally assigning user defined values). Basically, an enumeration type *e* denotes a set of symbols (sub-types) and an enumeration object *E* with properties (S_1,S_2,..). Consequently, `enum` declares an object and defines a type!

Optionally, the symbol value type (*symboltype*) and the enumeration tag (*ETag*, string prefix) can be declared, too. In the *JS* example below this is the string type and the '`E`' string prefix. An enumeration symbol can be a constant value (number or string, i.e., representing the tag) or a constructor function returning an object with a tag field, shown in the type definition templates below.

`enum` *E* `= {`
  S_1, *Abstract symbol*
  S_2,
  S_3 `=` *expr*, *User defined symbol with concrete value assignment*
  ..
} *[: e] [(symboltype [ETag])]*
`=` S_1 | S_2 | ..

`typeof` *E* `=` *e* `=` S_1 | S_2 | .. ⇔
`typeof` *E* `=` *e* `=` `'S.S1'` | `'S.S2'` | ..

$$\frac{\text{enum } S = S_i \mid S_j{=}v_j \mid ..}{\text{var } S = \{S_i : \text{'}S.Si\text{'}, S_j : v_j, .., \}} \tag{2}$$

Class Types, Implementations, and Objects

In *JS*, an object can be a traditional procedural record structure (with functions as attribute values) or an object associated with object-oriented (OO) programming defining private data and methods accessing the object data. Records are created on the fly and their type interface is particular for each object and have to defined for each object separately, i.e., `object` *o* `= {` e_1`:`typ_1`, .. }`. A real object in sense of OO defines a class from that objects can be instantiated. Additionally, a class defines data and methods (prototypes) that can access the data by using the `this` reference. There is a constructor function that instantiates an object with a given set of parameters. The parameters can be assigned to object attributes. Each class type definition creates a constructor function that can be used to instantiate objects from the class. Per convention the class type name starts with a lower case letter and the constructor function name starts with the respective upper case letter.

The transformation rule *JS+* $\rightarrow$*L*(*JS*) for a class implementation definition is shown in Eq. **3**. A *JS+* class type definition can be either a type declaration or a type declaration mixed with implementation, shown in the following code templates.

Interface [+Implementation]
`class` *t* `= {`
  Object Data
  e_1`:`typ_1 *[=val],*
  e_2`:`typ_2`,`
 `+`e_3`:`typ_3 *Extension*

  Methods
  m_1`:method (`p_1`:`$ptyp_1$`,`p_2`:`$ptyp_2$`,..)` $\rightarrow$ *typ [{statements}],*
  m_2`:method (`p_1`:`$ptyp_1$`,`p_2`:`$ptyp_2$`,..)` $\rightarrow$ *typ [{statements}],*
  `..`
`}`

Parameterized Class (providing constructor interface)
`class` *t*`(`p_1`:`$ptyp_1$`,`p_2`:`$ptyp_2$`,..) = {` e_1`=`p_1`,`p_2`,`e_3`:`typ_3`,.. }`

Constructor Function
`constructor` *T*`(`p_1`:`$ptyp_1$`,`p_2`:`$ptyp_2$`,..)` $\rightarrow$ *t* $\Leftrightarrow$ `constructor` *T*`({`p_1`:`$ptyp_1$`,`p_2`:`$ptyp_2$`,..})` $\rightarrow$ *t*
`object` *o*`:`*t*

$$\frac{\begin{array}{l}\text{class c}(p_1{:}ptyp_1,..) = \{\ e_1{:}typ_1, p_1, m_1{:}\text{method}(p_1{:}ptyp_1,,.)\{stmts\},..\ \} \\ \quad \text{function } C(p_1{:}ptyp_1,..)\ \{\text{this}.e1{=}\partial(typ_1);\ \text{this}.p_1{=}p_1;\ ..\} \\ \quad \text{C.prototype}.m_1{=}\text{function}(p_1{:}ptyp_1,,.)\ \{stmts\}\end{array}}{} \tag{3}$$

3.2. Distributed and Parallel Programming Types

The parallel *JS* programming model denoted by *JST* is derived from *ConPro* model. *ConPro* is a parallel programming language based on the Concurrent Communicating Sequential Process model (CCSP) and a synthesis framework for SoC hardware design (digital logic). Details about the *ConPro* programming language and the High-level synthesis framework can be found in **[6]**. The *Conpro* programming model is composed of communicating sequential processes (CSP, Hoare) extended with guarded concurrent access of shared resources (resolved by a mutex scheduler). The *ConPro* programming model can be synthesized to software (using threads and software processes)

and hardware (System-on-Chip designs, pure RTL, *VHDL*). Some transformation rules *JS+* →*L*(*ConPro*) are shown in Eq. **4** and the following code templates demonstrate the mapping of a process model on *JS*. Note that a *ConPro* process definition statements (except process arrays) create only one specific process, in contrast to *JS* object constructors that can create multiple instances each with its own set of parameters.

The design of hardware and digital logic requires bit-scaling of data objects. This is supported by *JST*/*JS+* with range constraint types: *type*(*size*), *type*(*from*,*to*), e.g., `integer(8)`, `number(0.0,1.0)`, ..

Software Process

Process Constructor Function

```
process constructor function pro(par1,par2,..) {
  statements
}
pro: constructor function(par1:t1,par2:t2,..) → process
```

Generic Process (Super) Class

```
process class = {
  start: method,
  stop: method,
  call: method(arg1:t1, arg2:t2, ..),
  status: method() → 'run'|'waiting'|'end'|'blocked'|'killed'|..
}
```

Definition of a Process Cluster with node placement and process assignment

```
cluster object cl = {
  nodei:[prok,pro1,..],..
}
```

process constructor function $pro(pa_1{:}ptyp_1,pa_2{:}ptyp_1,..)$ {*stmts*}
var $pro_1 = pro(v_{1,1},..), pro_2 = pro(v_{1,2},..)$; p_1.start();
var r_1:register integer(*width*);
---
process *pro*1() begin reg pa_1: τ $(ptyp_1)=v_{1,1}$; .. *stmts* end; (4)
process *pro*2() begin reg pa_1: τ $(ptyp_1)=v_{1,2}$; .. *stmts* end;
reg r_1:int[*width*];

3.3. Agent Model Types

The agent behaviour and programming model exists already (*AAPL* **[10]**) and is used in recent scientific research. It is based on an Activity-Transition-Graph (ATG) model composing the agent behaviour of activities and (conditional) transitions between activities. Activities perform actions: Computation and modification of agent body variables; Input- and Output operations (tuple space data base access, signals); Migration (mobility of agent processes), agent control (creation and destruction of agents, modification of ATG behaviour, ..). There are actually two different existing agent processing platforms based on the *AAPL* model: The JavaScript Agent Machine (*JAM*); and the *AgentFORTH* machine (*AFVM*). The *JAM* platform (entirely written in *JS*, too) do not need further transformation of the agent behaviour constructor. The creation and processing of agents is entirely handled by *JAM's* Agent Input and Output System (*AIOS*). Details can be found in **[7]**. The *AgentFORTH* machine processes machine instructions that require a compilation of the *JS* code. Details about *AFVM* and *AgentFORTH* can be found in **[8][9]**. Regardless of the target agent code and platform, *JS+*/*JST* enable the implementation of agents using a common programming language. An agent class template is shown below.

```
Definition of Agent Class Constructor
agent constructor function ac(par1,par2,..) {
  Body Variables and aux. Functions
  this.v1=expr;
  this.v2:typ=expr;
  this.f=function (..) { .. }
  ..
  Activities
  this.act = {
    a1: function (..) { .. },
    a2: function (..) { .. }, ..
  }
  Transitions
  this.trans = {
    a1: an, ..
    ai: function () { .. },
  }
  Signal Handler
  this.on = {
    S: function () {..},
    ..
  }
  this.next=a1;
}
```

The agent behaviour is modelled and specified with a structural name-semantic convention in mind. The respective semantic type overlay of the agent class constructor is shown below associating objects attributes defined by the constructor function with derived semantics and finally a generic type interface:

```
type agent constructor => function (parameters) → agent class
type agent class = {
  @identifier : body attribute,
  act :         activity {},
  trans :       transition {},
  on:           signal handler function {},
  next:         string,
}
type activity   => subclass process => function;
type transition => activity → activity => string|function () → string
```

3.4. Abstract Data Types

Graphs and Trees

The template type for generic graphs and trees are shown below. A graph can be defined with a class specifying the elements of the graph (nodes/vertices and edges) and a network object specifying the structure. Nodes and edges are distinguished by *node* and *edge* classes. The internal graph structure (data organization) is hidden. There is a common set of methods used for constructing and modifying graphs (e.g., *insert, connect, disconnect, remove,* ..);

```
Graph Type Definition
graph class G = {
  nc1:node class,   Abstract class
  nc2:node class { ei:ti,mj:method,..},   Concrete class
  ec1:edge class,
  ..
}
```

```
Generic Graph (Super) Class
graph class = {
  insert: method,
  remove: method,
  connect: method,
  disconnect: method,
```

```
  search: method,
  path: method
}
```

Graph Structure Definition

```
graph object g = {
  Compact Form
  obj1: node typ1 → obji with edge:e,
  obj2: node typ2 → [obji,objj,..] with edge:e,

  Alternatively
  obj1: node typ1,
  obj2: node typ2, ..
  edg1: edge directed typ1 → [obji,objj,..], ..
  ..
}
```

Node and Edge Class Definitions

```
node class n = { ei:ti,mj:method,..}
edge class e = { ei:ti,mj:method,ck:constraint function,..}
```

3.5. Physical Model Types

The physical domain is closely related to simulation, although it is possible to define physical design models that can be used in manufacturing and mechanical design of a broad range of parts and structures.

Multi-body Physics

A multi-body physics model is a network graph of mass nodes with edges of springs. A constraint function can be, for example, the hooks law applied to all springs. Concrete parts of a model can be defined by a `part class`, more generic physics models can be defined by a `model class`, shown in the following *JS*+ type template.

```
physics part class P = {
  mass: mass class {m:number,..},    Generic mass node class
  massi: mass object {m:number=..,..},    Specific mass node object
  spring: spring class {k:number,l0:number,constraint:function,..},
  springj: spring object {k:number,l0:number,constraint:function,..},
  connect: link function (obj1:mass object, obj2:mass object) {..} → spring object,
  network: [mass object,mass object, spring object] [] = [..],
  ..
}
physics model class M = {
  mass: mass constructor function () → mass,
  spring: spring constructor function (k:number,l0:number,..) → spring,
  constri: constraint function (x,y,..) → z,
  ..
}
```

3.6. Hardware Model Types

Hardware design, i.e., the design of digital logic and System-on-Chip microcomputers using *VHDL* is a challenge and differ from software development significantly. Hardware design requires the specification of a hardware behaviour model, commonly using hardware description languages like *VHDL* or *Verilog*. The *JST* hardware model is closely related to *VHDL*. The most relevant part supporting hardware design with *JS*/*JST* are bit-scalable types *type*[*size*] and the concept of signals (type `signal`). A hardware component (i.e. *VHDL* entity and architecture) is represented by an object class (type `hardware component`). Hardware processes are represented by process functions in *JST* that are methods of this component object that can access port and internal signals by using the `this` reference as shown in the type template below. Block functions represent top-level statements of the *VHDL* architecture. They have to be pure combinatorial. A principle transformation pattern *JS*+ → *VHDL* is shown in Eq **5** and the generic form of a hardware component class template is following.

```
hardware component class M = {
  port: {
    p1: input signal logic[n],
    p2: output signal logic[n],
    clk: input signal std_logic,
    ..
  },
  body: {
    s1: signal logic[n],
    s2: signal number,
    t1: type ..,
    pr1: process function (){ this.s1=0; if (this.clk.event(1)) {..} },
    pr2: process function,
    b1: block function () { this.p2=this.s1==0?this.p1:0 },
    ..
  }
}
hardware process function pr2() {
  statements
}
```

$$\frac{\begin{array}{l}\text{hardware component class } M = \{\\ \text{port: } \{ p_1\text{:input signal } tp_1, ..\}, \text{ body: } \{s_1\text{:signal } ts_2, pr_1\text{:process,..}\}\end{array}}{\begin{array}{l}\text{entity } MOD\_M \text{ is port(signal } p1\text{: } \tau\ (tp1),..) ..\\ \text{architecture } A \text{ of } MOD\_M \text{ is signal } s_1\text{: } \tau\ (ts_2),..\text{begin } pr_1\text{: process(}\ \sigma\ \text{) begin ..}\end{array}} \quad (5)$$

3.7. Communication and Network Types

Networks

One major model class is a network, i.e., a sensor or communication network. Network models can be generic just defining the structure of the network consisting of nodes (defined by the *node class*) equal to the graph class model or concrete with communication links (attached to communication ports defined by the *link class*) addressing information communication. Links are attached to ports defined by the *links* class using message passing defined by the *proto class* (optional). A node can supply an arbitrary set of ports. A communication link can be modelled on different abstraction levels defined by the *link class*. A link can be associated with a communication protocol or a specific hardware communication device. A node can implement a service loop managing messages and time-outs (defined by the *service* handler). The components of a network (nodes, ports, links) can be specified inlined in the network class definition or separately, shown below.

Network Type Declaration and Definition [with optional implementation]

```
network class N (i:number,j:number,..) {
  User defined components
  node: network node {..},
  port: network port { .. },
  link: network link {..},

  links: network link [],
  Combinator creating connectivity
  autoconnect?: function (network node,network node) → boolean|port [],
  Service handler
  service?: function (event:string,arg:*) { .. }
}
```

Generic Network (Super) Class

```
network class = {
  insert: method (node),
  remove: method (node),
  connect: method (from:node|port,to:node|port|(node|port)[]) → link
  disconnect: method (from:node|port,to:node|port|(node|port)[]),
  path: method (node|port) → link [],
}
```

Network Structure Definition

```
network object Nx = {
  obj1:node object typ1 → obji with link:l,
  obj2:node object typ2 → [obji,objj:pt2,..] with link:l,
  ..
}
node class n = {
  service?: function (options) { .. },
  sensors: sensor class [],
  actuators: actuator class [],
  ports:{pt1,pt2,..}, ..
}
port class p = {
  node: node class,
  link: link class|link class []|undefined,
  read: function () {..} → data:*,
  write: function (data:*) {..},
  stats: function () {..} →  number,
  proto: proto,
  options : {delay,bandwidth,..}
}
link class l(p,..) = {
  from:network port,
  to: network port,
  read: function () {..} → data:*,
  write: function (data:*) {..},
  stats: function () {..} →  stats,
  proto: proto|proto [],
  options : {delay:number,bandwidth:number,..}
}
proto class = {
  message : {
   ..
  }
  ..
}
```

3.8. Input-Ouput

Input and output classes can define sensors, actuator, files, network interfaces, or data bases. Some class templates for sensors and actuators are shown below.

```
sensor class s (options) = {
  kind:string,
  data: β,
  model: method (x:β) → α,
  read: method () → α,
  init: method,
  calibrate: method ([α,β][]),
  ..
}
```

```
actuator class a (options) = {
  kind:string,
  data: β,
  read: method () → α,
  write: method (α),
  init: method,
  model: method (x:β) → α,
  calibrate: method ([α,β][]),
  update: method,
  ..
}
```

4. The Tools and Development Framework

4.1. The System Compiler (SYCO)

The entire tool framework is composed of multiple modules. All parts and tools of the compiler framework (except external tools like *ConPro* or *Abaqus*) are written in *JS*! The most central parts are the *JS*/*JST*/*JS+* parser *estprima* and the *JS* code generator *estcodegen*. The input model and program in *JS+* format is parsed by *estprima* to an abstract syntax tree (AST) that is analyzed with rules from a model data base. The AST is further processed by different compiler and back-end modules producing a wide variety of output formats.

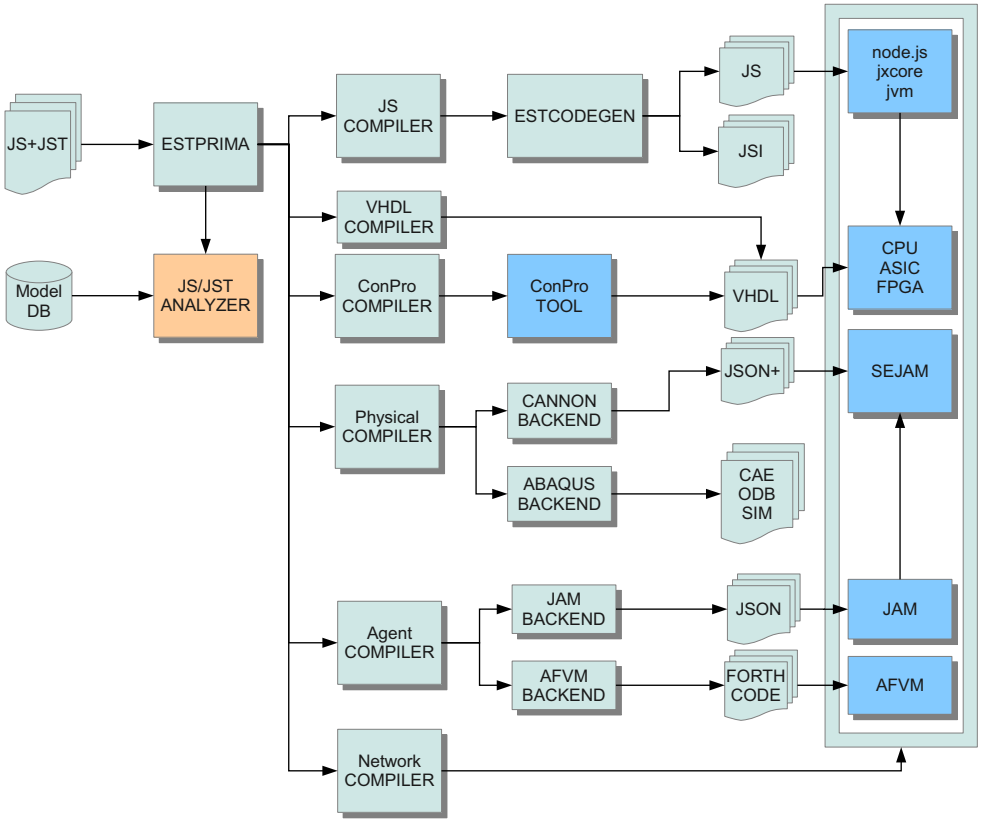

Fig. 4. The System Compiler Framework using one unified input language and meta model to synthesize a system design and configuration. All parts and tools of the compiler framework (except external tools like *ConPro* or *Abaqus*) are written in *JS*! The most central parts are the *JS*/*JST*/*JS+* parser *estprima* and the *JS* code generator *estcodegen*.

4.2. Agent Platforms: JAM and AFVM

The JavaScript Agent Machine (*JAM*, Fig. **5**, right side) **[7]** is capable of executing a large number of mobile agents directly based on the agent model introduced in Sec. 3.3. (*AgentJS* based on the *AAPL* meta language **[10]**). Mobile agents are used to perform data processing on the three layers of a sensor and control system: Sensing; Aggregation; Application (Control). *JAM* can be deployed in strong heterogeneous environments ranging from miniaturized sensor networks (smart materials) to the Internet and Clouds.

The agent code (represented by an agent process encapsulating code and data) is capable of migration in networks between nodes as a requirement for distributed data processing.

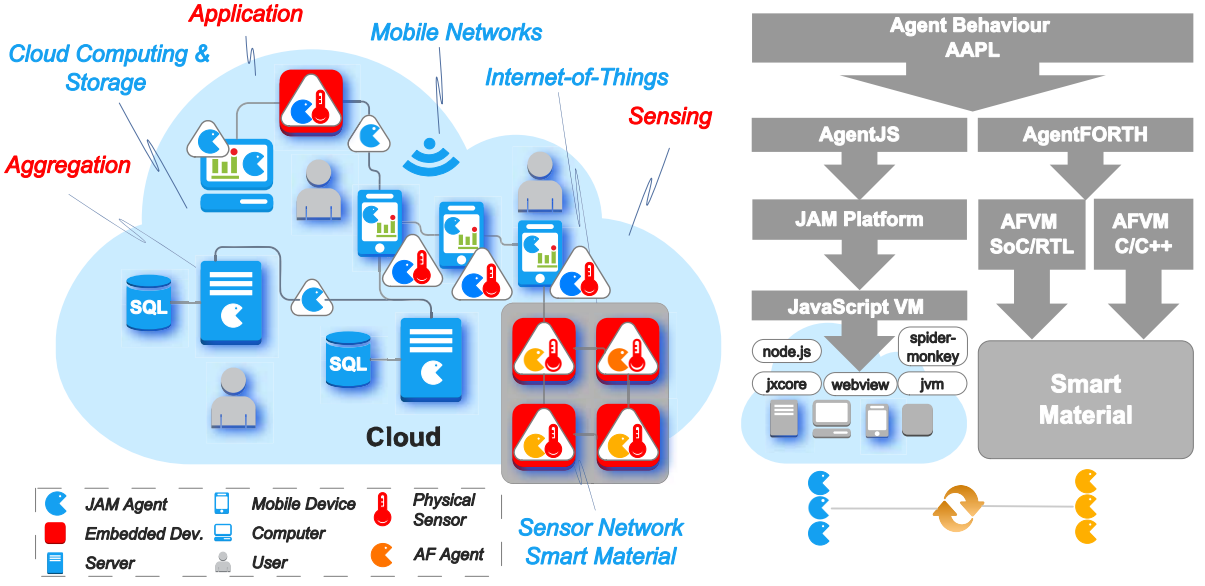

Fig. 5. Unified MN/IoT/ Cloud Distributed Perception and Information Processing with mobile agents using a hybrid platform framework consisting of the JavaScript Agent Machine Platform (JAM) and the low-resource *AgentFORTH* machine (AFVM)

For low-resource networks, a stack processor based approach (with *AgentFORTH* code) is used instead (Fig. **5**, right side) featuring hardware implementation and advanced token-based agent process scheduling.

4.3. SEJAM: Multi-domain Simulator

SEJAM is a multi-domain simulator that combines MAS and physics simulation in one monolithic program. The simulator already benefits from the *JST*/*JS* programming and modelling capabilities (Modules Agent Model, Communication Model, and Physics Model).

The coupled physical and computational simulation consists of two engines:

- Physics: Multi-body physics solver using a meshgrid network of nodes cnnected by damped springs that can be parameterized; and
- Computation: Mobile Multi-Agent Systems and Networks of JavaScript Agent Machines processing agents.

It features a fully JavaScript based modelling and programming environment, and SEJAM2 is programmed entirely in JavaScript, too! Agents are deployed in ICT networks:

- Each node of the network is coupled to sensors and actuators
- Agents can access sensors and control actuators
- Sensors and actuators are modelled with multi-body physical systems
- Direct interaction between agents and physical system and vice versa

A snapshot of a simulation run with a plate consisting of a 8x5x3 network of computer and mass nodes connected by springs is shown in Fig. **6**. The objective of the complex simulation is to investigate the interaction of computational systems (agents) with sensors, actuators, and physical systems.

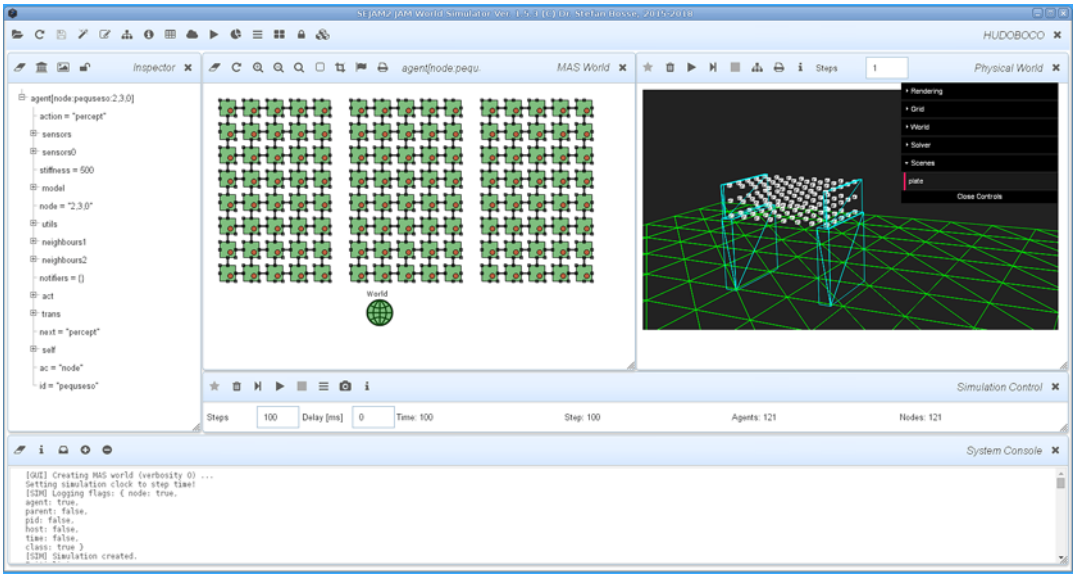

Fig. 6. *SEJAM2* Simulator with a combined MAS (center) and MBP simulation (right) based on *JST*/*JS*/*JSON* modelling.

The simulation model containing both physical and computational parts is specified with a structural name-semantic notation and an overlayed type signature:

Pure JavaScript Notation

```
simulation : model = {
  // Physical MBP model
  physics : { plate : {..}},
  // Computation and Communication: Agent behaviour Classes
  classes: {
    node   : { behaviour: function () {..}, visiual:{..} },
    broker : { behaviour: function () {..}, visiual:{..} },
    notify : { behaviour: function () {..}, visiual:{..} },
    world  : { behaviour: function () {..}, visiual:{..} },
  },
  // Global simulation parameters used by agent and physical simu.
  parameter : {
    strainDelta: 0.1, optimizaton: 'segment',
    stepPhy: 100,     stiffness: 500,
    holes: [[1,2,0],[1,2,1],[1,2,2], .. ],
    ..
  },
  // Simulation set-up and initialization
```

```
  world: { ..
    init : { agents: {..}, physics: {..}},
    meshgrid : { node:{..}, port = {..},
                 link:{..} .. } } // Network model
}
⇒
Semantic Type Model
type model = {
  physics :  multibody component class {},
  classes :  agent descriptor {},
  paramater: simulation parameter {},
  world : {
    init : fucntion {},
    meshgrid : world class,
    ..
}
type world class = {
  node: node class, port:port class,
  link: link class}
}
type agent descriptor = {
  behaviour: agent constructor,
  visual : visu
```

5. Conclusion and Outlook

The design and simulation of complex sensing, mechatronic, and intelligent systems require a unified system modelling and programming language. This work introduced JavaScript *JS+* with a semantic type system extension as a possible solution to fill the gap between models and implementations. The advantage of this approach is the provision of a unique and all-round development environment with only one input format and producing multiple output formats suitable for non-experts. Since *JS* can be processed by *JS* the development of new tools is simplified. The interface to the new *JS+* language is provided by the (already existing) *estprima* parser transforming source text in an AST and the *estcodegen* code generator producing generic *JS* and type interfaces *JST* from this AST. Some tools exist already that profits from the new *JS+* modelling approach like the multi-domain simulator *SEJAM* that is used to simulate complex mechatronic or structural monitoring systems and the interaction of distributed computational with physical systems.

More domains and models have to be implemented in *estprima* and *estcodegen*. More over, complex case studies have to be evaluated leading to a refinement of *JS+* semantics and tools for a high practical and expressive benefit.

6. References

[1] J. Stoppe. R. Drechsler, *Analyzing SystemC Designs: SystemC Analysis Approaches for Varying Applications*, Sensors MDPI, pp. 10399-10421, 2015.

[2] H. Al-Junaid, T. Kazmierski, and L. Wang, *SystemC-A Modeling of an Automotive Seating Vibration Isolation System*, in Forum on Specification and Design Languages (FDL 2006), Germany. 19 - 22 Sep 2006, 2006.

[3] Chapuis, Y.-A & Zhou, L & Fujita, Hiroyuki & HervÃ©, Y. (2008). *Multi-domain simulation using VHDL-AMS for distributed MEMS in functional environment: Case of a 2D air-jet micromanipulator*. Sensors and Actuators A: Physical. 148. 224-238. 10.1016/j.sna.2008.07.025.

[4] T.R. Egel, N.J. Elias, *Using VHDL-AMS as a Unifying Technology for HW/SW Co-verification of Embedded Mechatronic Systems*, SAE 2004 World Congress.

[5] M A McEvoy, Nikolaus Correll, *Materials science. Materials that couple sensing, actuation, computation, and communication*, Science, 347(6228):1261689 (2015)

[6] S. Bosse, *Hardware-Software-Co-Design of Parallel and Distributed Systems Using a unique Behavioural Programming and Multi-Process Model with High-Level Synthesis*, Proceedings of the SPIE Microtechnologies 2011 Conference, 18.4.-20.4.2011, Prague, Session EMT 102 VLSI Circuits and Systems, 2011, DOI:10.1117/12.888122.

[7] S. Bosse, *Mobile Multi-Agent Systems for the Internet-of-Things and Clouds using the JavaScript Agent Machine Platform and Machine Learning as a Service*, in The IEEE 4th International Conference on Future Internet of Things and Cloud, 22-24 August 2016, Vienna, Austria, 2016

[8] S. Bosse, *Unified Distributed Computing and Co-ordination in Pervasive/Ubiquitous Networks with Mobile Multi-Agent Systems using a Modular and Portable Agent Code Processing Platform*, in The Proc. of the 6th EUSPN 2015, Procedia Computer Science

[9] S. Bosse, *Design and Simulation of Material-Integrated Distributed Sensor Processing with a Code-Based Agent Platform and Mobile Multi-Agent Systems*, Sensors (MDPI), 15 (2), pp. 4513-4549, 2015, DOI:10.3390/s150204513.

[10] S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, IEEE Sensors Journal, Special Issue MIS, 2014, DOI:10.1109/JSEN.2014.2301938.

Appendix A. JS/JST/JS+ Examples

Typed Software Program

```
type sensors =
  Strain {x:number,y:number:z:number} |
  Temperature { degree:nummber, kelvin:number } |
  Voltage {v:number }

var v:sensors = sensors.Voltage(100),
    s:sensors = sensors.Strain(0,0,0);

function sense(s:sensors) {
  switch (s.tag) {
    case sensors.strain:      return s.x+s.y+s.z;
    case sensors.temperature: return s.kelvin;
  }
}
class controller (options:{k:number,x0:number}) {
  k=options.k,
  x0=options.x0||0,
  p:method (x:number) -> number { return (x-this.x0)*this.k },
  set: method (x) -> number { this.x0=x }
}
var pid = new Controller({k:10});
```

Sensor Network

```
sensor class T(options) = {
  data:number,
  read: function () { return this.data*this.options.k }
}
network class SN = {
  node: network node class = {
    id:string,
    pos:{x:number,y:number},
    ports: network port []=[this.port(NORTH),this.port(SOUTH),..],
    sensors: T [],
    init: function () {
     this.sensors=[T(k:1)]
    }
    service: function (event,arg) {
      switch (event) {
        case 'timeout': return 1000;
        case 'message': ..
        case 'signal': ..
      }
    }
  },
  port: network port class = {
    dir:string,
```

```
      read:function () {..},
      write: function(data) {..},
    }
    link: network link class = {
      from: network port,
      to:   network port
    },
    autoconnect: function (n1:network node,n2:network node) {
      return (Math.abs(n1.pos.x-n2.pos.x)==1 || Math.abs(n1.pos.y-n2.pos.y)==1)
    }
  }
```

Physical MBP Plate Model (for Simulation)

```
physics type element = mass {} | spring {}
physics model class plate(i:number,j:number) = {
  ms1: mass constructor function (u,v) {
    return mass({m:100,unit:'g',x:u*5,y:m*5,dx:5,dy:5,visual:{color:'red',shape:'box'}})
  },
  sp1: spring constructor function (n1,n2) {
    return spring({k:50,constraint:constraint function (s) { return s*this.k } → force})
  },
  create: part constructor function () {
    var elements={};
    for(var y=0;x<this.j;y++) for (var x=0;x<this.i;x++) {
     var node=[x,y].join(','),prev=x>0?[x-1,y].join(','):null;
     elements[node]=this.ms1(x,y);
     if (prev) elements.push(this.sp1(elements[prev],elements[node]));
    }
    return elements;
  }
}
```

Agent Behaviour Model

```
agent constructor function explorer(range:number,data:*) {
  this.range=range;
  this.distance=0;
  this.data=data;
  this.dir:dir=null;
  this.act = {
    migrate: function () {
      var dir=random([DIR.NORTH,DIR.SOUTH,DIR.EAST,DIR.WEST]);
      if (link(dir)) moveto(dir);
    },
    deliver: function () {
      out(['SENSOR',this.data])
    }
  }
  this.trans = {
    migrate: function () { return this.distance==this.range?deliver:migrate }
  }
  this.next=migrate
}
```

Multiprocess Model

```
open process;
open system;
open ipc;
var ev:event object=Event();
var ar:array(256) integer(8) object=Array(256);
```

```
process function main() {
  worker1.start();
  worker2.start();
  ev.wakeup();
}
process function worker1() {
  ev.await();
  for (var i=0;i<128;i++)
    ar[i]=0;
}
process function worker2() {
  var r:register integer(8)=0;
  ev.await();
  for (var i=128;i<256;i++)
    ar[i]=0;
}
```