A Unified System Modelling and Programming Language

Based on JavaScript and a Semantic Type System

PD Dr. Stefan Bosse
University of Koblenz-Landau, Institute of Software Technology
19.6.2018
sbosse@uni-bremen.de

1. Inhalt

2. Introduction and Overview

Central Question. *How to model, design, simulate, and implement (program) complex Cyber-Physical Systems with one unified approach and language?*

2.1. Design of Complex Systems

- The **design and modelling of complex and heterogeneous distributed sensing and physical control systems** is a **challenge**!
- 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;
  - Hardware;
  - Software using different modelling and programming languages.

2.2. Modelling and Programming

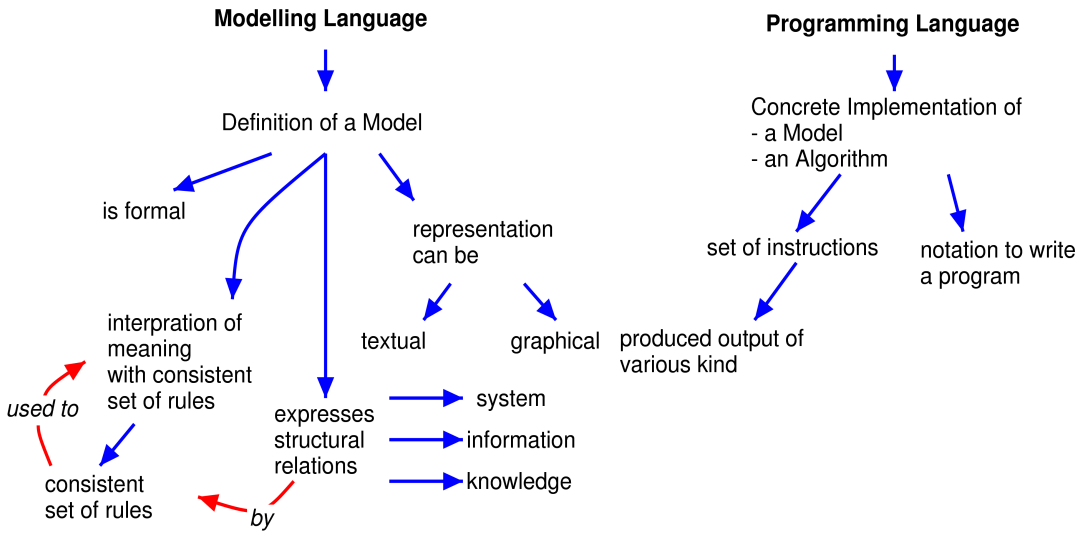


Fig. 1. Difference between Modelling and Programming Languages

2.3. Design and Simulation

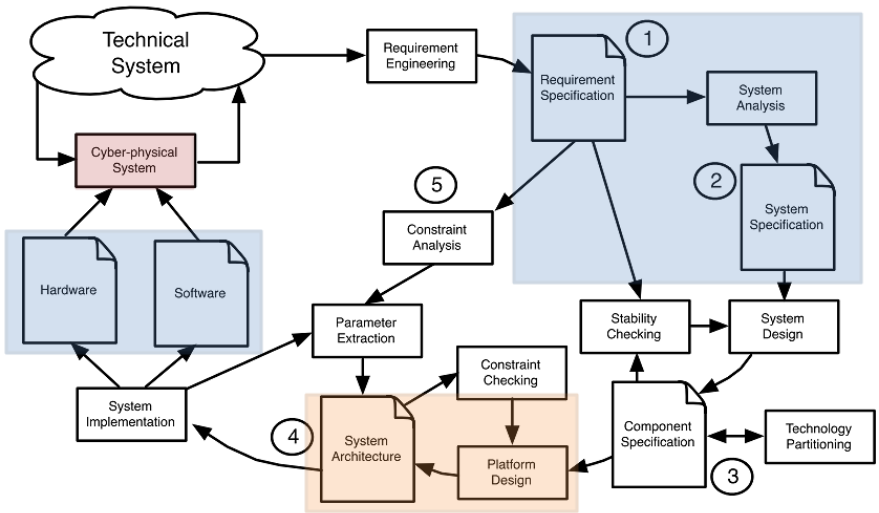


Fig. 2. Overview of the design flow of Cyber-Physical-Systems [1]

2.4. Design Domains

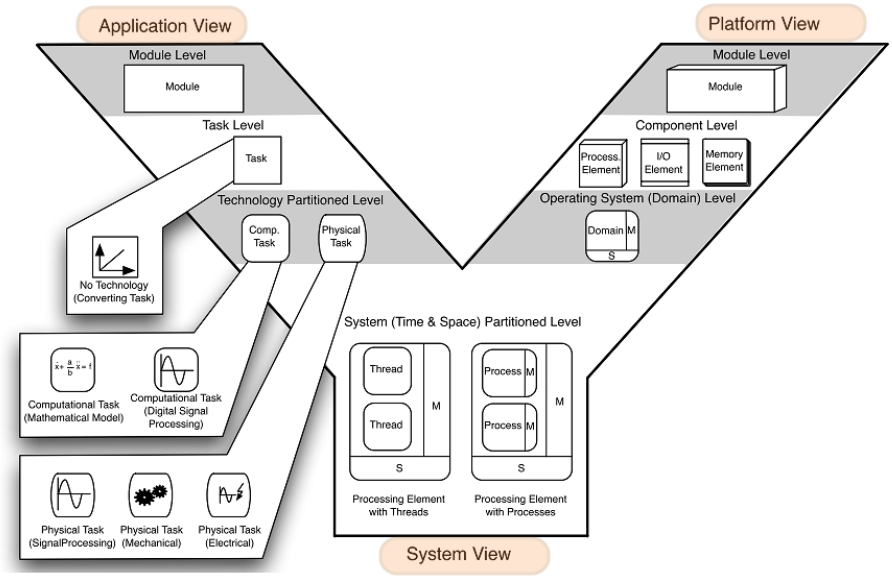


Fig. 3. There are different abstraction layers in the design flow and three design domains [1]

2.5. Example: Sensorial and Adaptive Material

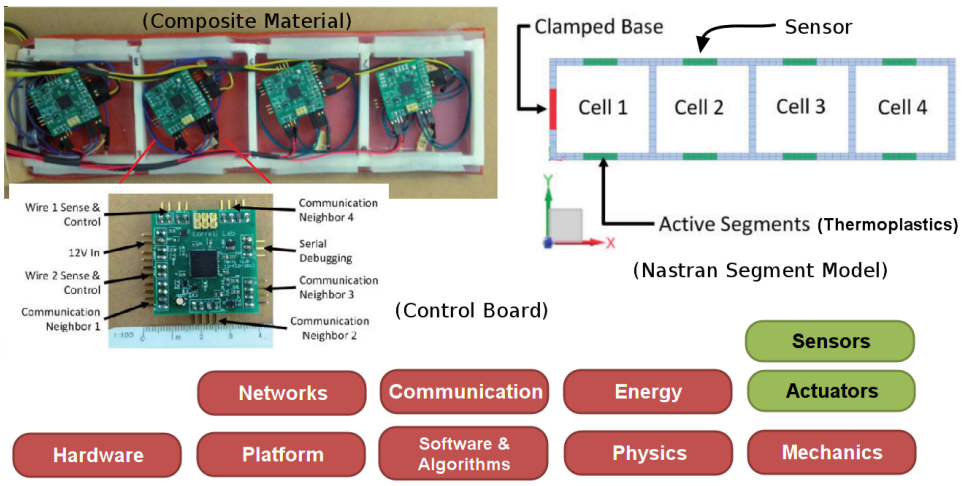


Fig. 4. An example of a sensorial and adaptive material consisting of sensors, actuators, and control nodes [2]

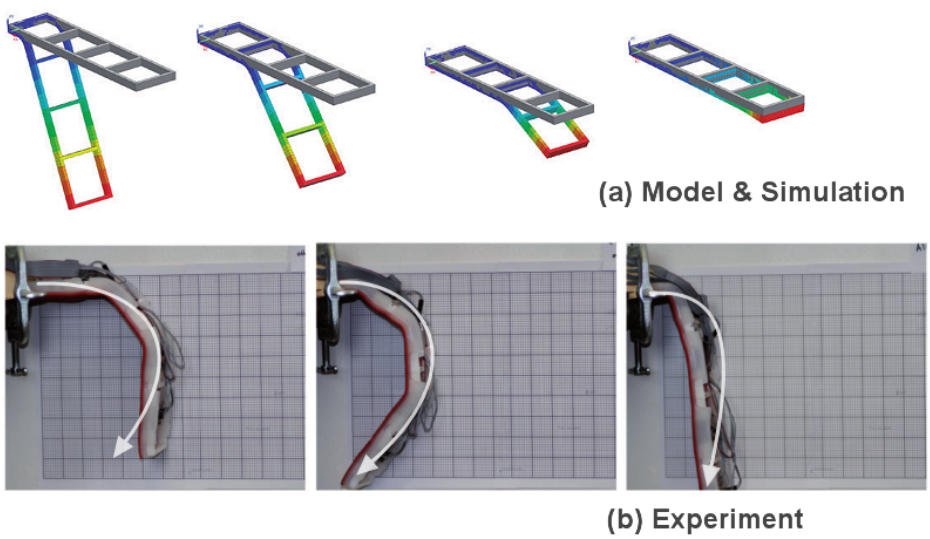


Fig. 5. Different geometric shapes of the structures caused by different temperature distributions of thermoplastic actuators - The precise control and outcome of geometric shape change is difficult to perform [2]

2.6. System Meta Modelling Language (SysML)

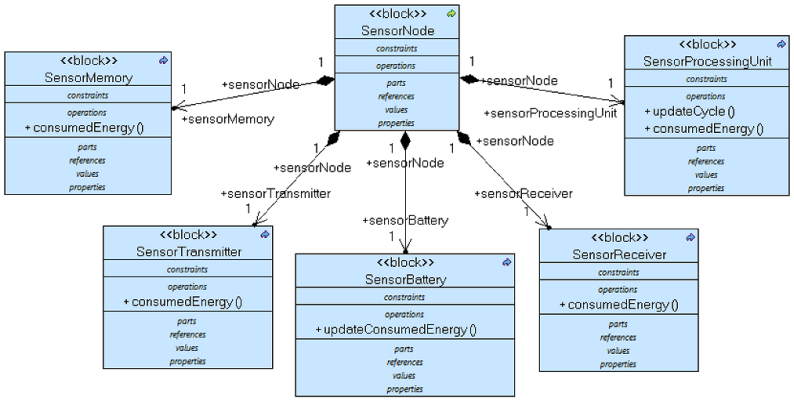


Fig. 6. Example of a structural SysML model of a smart sensor node composed of blocks (Platform & Architecture) [3]

2.7. Modelling with SystemC

Distributed Sensor Networks

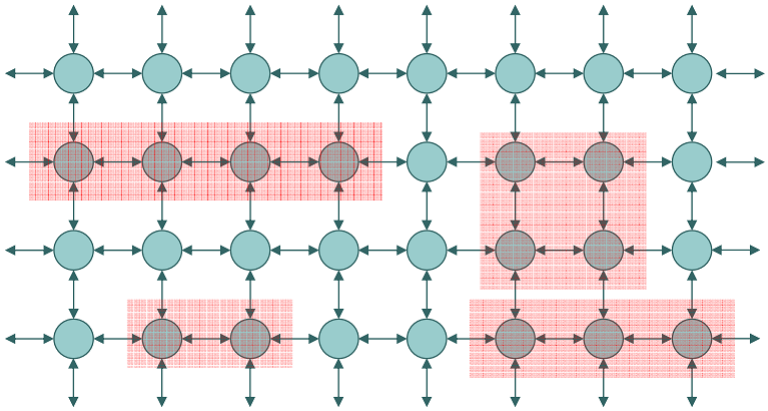


Fig. 7. Composition of a Distributed Sensor Network of core cells (sensor nodes) [4]

Smart Sensor Node

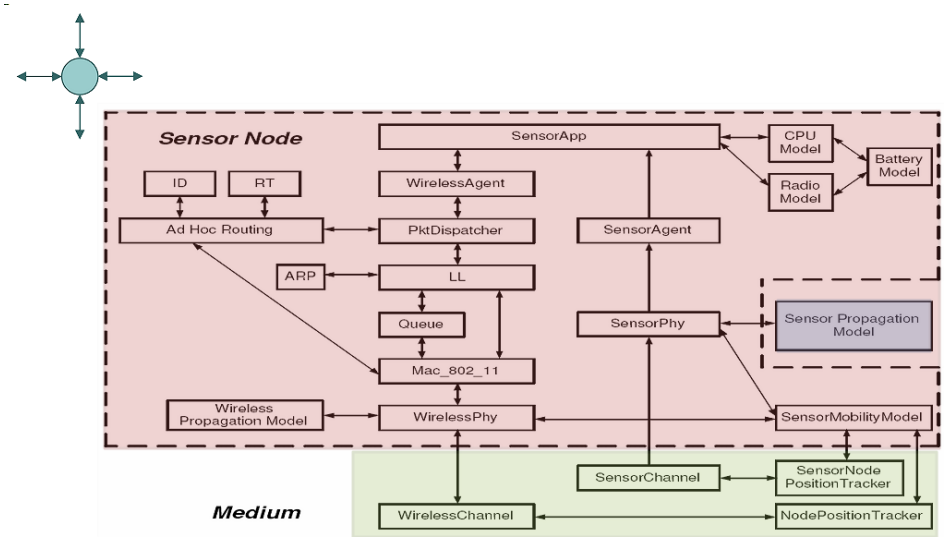


Fig. 8. SystemC Model diagram of one sensor node, the sensor model, and environmental medium (communication, physical, ..) [4]

Example: SystemC-AMS Model of an Air Temperature Sensor [6]

```
SC_MODULE(air_temp_sensor) // Air sensor model using ELN primitive modules
{
  sca_tdf::sca_in< double > in;
  sca_eln::sca_terminal out;
  sca_eln::sca_node_ref gnd;
 sca_eln::sca_tdf_isource *i1; // current source declaration
  sca_eln::sca_r *r1; // resistor declaration
  SC_CTOR(air_temp_sensor) // standard constructor
  {
   r1 = new sca_eln::sca_r ( "r1" ); // resistor instantiation
    r1->p ( out );
    r1->n ( gnd );
   i1 = new sca_eln::sca_tdf_isource ( "i1", 0.025 ); // current source Instantiation
    r1->value = 1; // R=1 ohm
    i1->p ( gnd );
    i1->n ( out ); // 0.0-3.25 (V)
    i1->inp ( in );
   }
};
```

2.8. Complex Design Tools

Hardware-Software Co-design

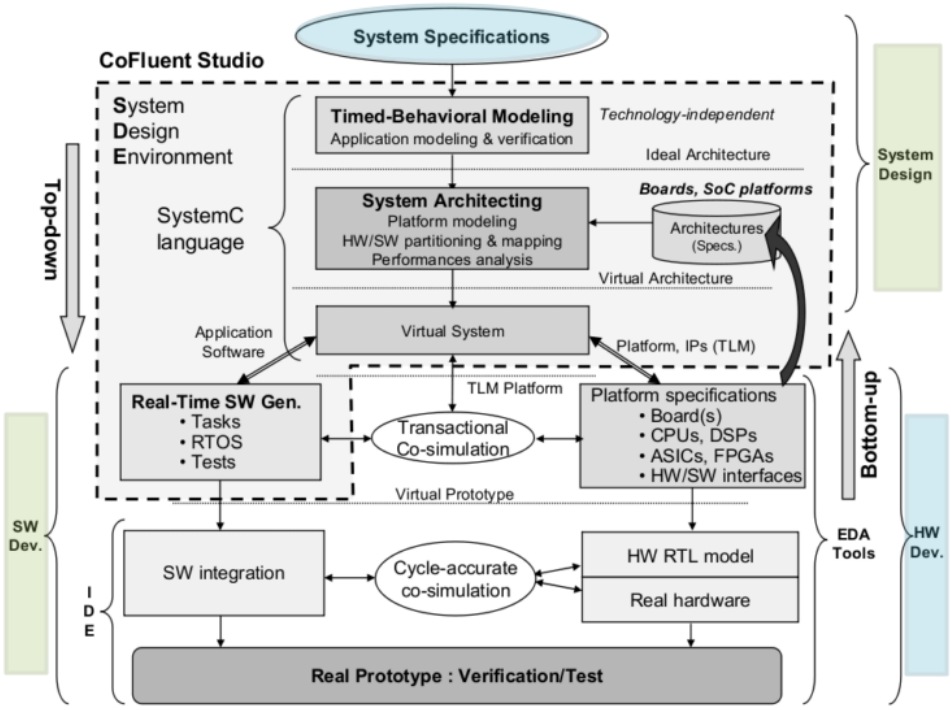


Fig. 9. CoFluent Studio: Embedded System design and synthesis of hardware, software, and systems using SystemC [5]

3. Model of Models

We have to distinguish
Implementation → *Programming, and*
Modelling → *Specification* → *Declaration*

System-of-Systems ⇒ *Model of Models!*

3.1. Modelling and Programming

- 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 → 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)
  - **Modelica** → Object-oriented, declarative, multi-domain modeling language for component-oriented modeling of complex systems
- All modelling and programming languages differ significantly with respect to syntax and semantic
- It is difficult and not very intuitive to bind all these different languages in one system-of-system design tool covering all levels and domains!
- A unified modelling and implementation language is require to cover all areas and domains!

3.2. The Concept

JavaScript for Modelling and Programming

Basic idea. Create a unified programming and modelling language for complex hardware-software systems (like sensor networks) by

- The (re)usage of **core JavaScript (JS)** and adding an
- **Extended semantic type system (JST) providing semantic relations**.

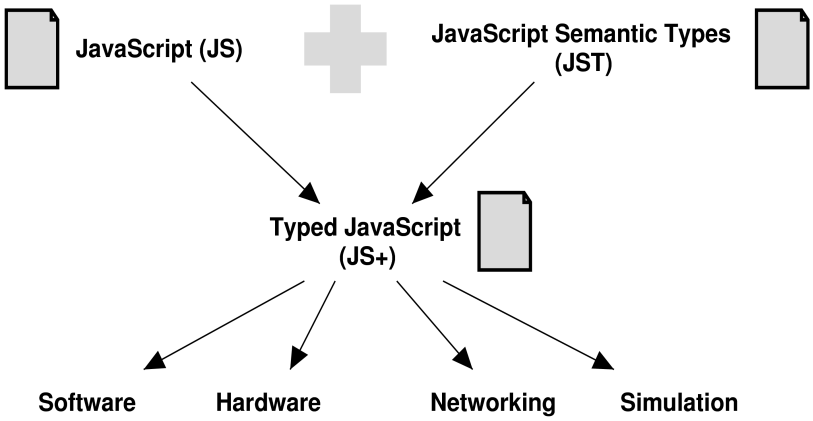


Fig. 10. The JS+JST=JS+ Concept

3.3. JavaScript

Why JavaScript?

- JavaScript is a **procedural-functional-object-orientated programming language** supporting dynamic / polymorphic typing at run-time
  - JS has no strong bindings to compilers or execution platforms (JS VM 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 and bound to any context.
  - Data and code can be **handled uniquely and exchanged** by the JSON+ file format (JSON extended with function code).
  - JS code and data can be **extended** at run-time
  - But **JS lacks of a type system**!
- The advantage of using one widely deployed 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**.

3.4. Hierarchical 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:

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

➤ Each domain consists of multiple elements (components) representing types in the semantic type system. The system model can be extended

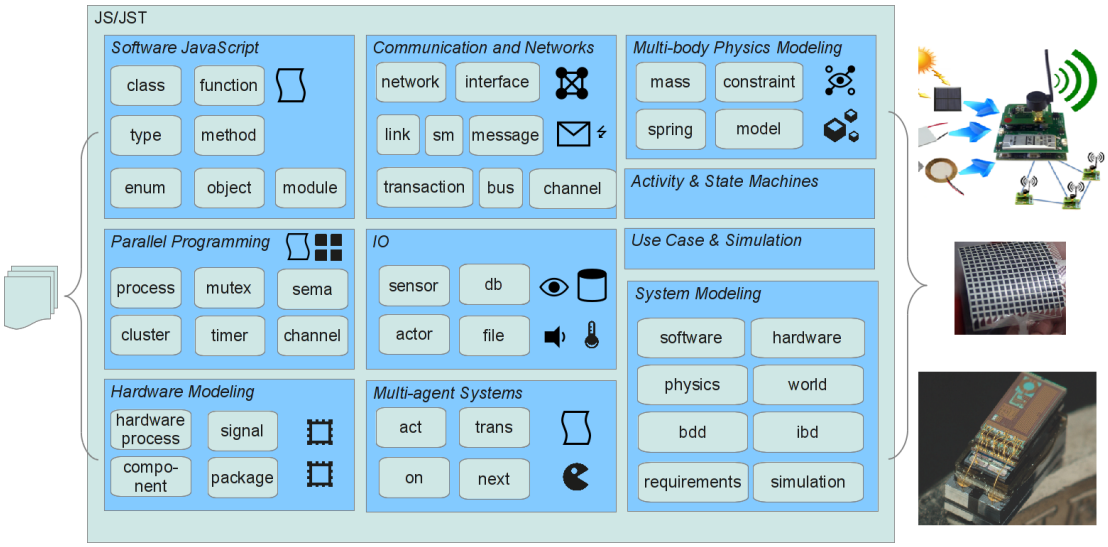


Fig. 11. System Model covering all domains by one unified modelling and programming language JS+/JST

3.5. Semantic Type System

➤ The central concepts of the JavaScript programming model are generic **functions** and generic **objects**.

- The Semantic Type System (JST) adds "semantic annotations" to functions and objects and introduces **type constraints** and **type signatures**
- Originally the JST was used to type constraint program code only (*software level*) and was specified separated from the implementation.

Examples of JS/JST Software Level

```
JS :  function foo(a,b,c) { return a+b+c }
JST:  foo: function (number,number,number) → number
JS :  var x,y,z;
JST:  type list; x: object, y: function [], z: list [];
JS :  function List (params) { this.params=params }
      List.prototype.method = function () { .. }
JST:  List: constructor function (params:{}) → list
      class list = { .. }
JS :  var Syms = { S1:'S1', S2:'S2'; ..}
JST:  enum Syms = S1 | S2 | .. : syms
```

4. The JavaScript Semantic Type System

4.1. Overview

- Association and definition of semantics by:

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

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

  2. 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 = { .. }
```

JS

- no type interface, only implementation

```
var o = {
 a: 1,
 b: 'text',
 c: function (p) {..},
}
```

JST

- Adds type signatures to implementations and type definitions as overlays

```
typeof o = {
 a: number,
 b: string,
 c: function (p:number []) {
    ..} -> number,
}
type DIR = NORTH | SOUTH |
           WEST | ..
```

JS+

- mixed implementation and type signature/definitions
- embeds types in implementations

```
var o = {
 a: number = 1,
 b: string = 'text',
 c: function (p:number []) {
    ..} -> number,
}
type DIR = NORTH | SOUTH |
           WEST | ..
var dir : DIR;
```

4.2. Software Level

Typed Objects

- In JS+ an object can be defined with an embedded type signature:

```
var obj-name = {
  e1: typ1 = val1,
  e2: typ2 = val2,
  ..
  en: typn = valn,
}
```

Classes

- Either separate class type interface and implementation or combined:

```
class class-name [ (par1,..) ] = {
  attr1: typa1 [ = val1 ],
  ..
  attrn: typan [ = valn ],
  m1: method (par1,..) [ { statements } ] → typm1,
  ..
}
```

- Each class type is associated with a constructor function

```
class t ⇔ constructor T (par1,..) → t
```

Product and Sum Types

```
type struct-name = {
  e1: typ1,
  e2: typ2,
  ..
  en: typn
}
type sumtype-name =
 S1 { tag=S.S1, .. } |
 S2 { tag=S.S2, .. } | ..
 Sn { tag=S.S3, .. }
```

4.3. Parallel Programming Level

- Parallel programming and behaviour models can be used to implement parallel software (for multi-processors) and parallel hardware systems (digital logic)
- Concurrent Communicating Sequential Processes (CCSP) using process constructors:

```
process constructor pro-name (par1,..) {
  statements
}
```

- Definition of process cluster and process placement on processor nodes

```
cluster object cluster-name = {
  node1: node placement object = [pro1,pro2,..],
  ..
}
```

4.4. Distributed Programming Level

Multi-agent Systems

- Reactive activity-based agents are modelled with Activity-Transition Graphs and a set of body variables
- Definition of an agent behaviour class via a constructor function with structural name-semantic convention:

```
agent constructor agent-class-name (par1,..) {
  Body Variables and private aux. Functions
  this.xi : typ2 = expression;
  Agent activities                Agent activity transitions
  this.act = {                      this.trans = {
     ai : function () { .. },          ai : aj,
    aj : function () { .. },        ak : function () { return al .. },
     ..                                  ..
  }                                 }
  Agent signal handler
  this.on = {
    Signal : function (arg,from) { .. }, ..
  }
  this.next = ai;
}
```

Multi-agent System: Model Type Signature

```
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: function;
type transition: string|function () → string
```

4.5. Network and Communication Level

- Communication models define communication ports, links, networks, and communication protocols
- Networks are composed of node and link classes
- Node classes contain computational classes, sensor classes, port classes

```
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:*) { .. }
}
```

4.6. Hardware Level

- Among data processing on hardware level (**digital logic**) the hardware level also defines *sensors*, *actuators*, and *analog electronic* circuits
- The hardware models define transfer functions (Input-Output relations) and constraints

```
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 },
    ..
  }
}
```

4.7. Physics Level

- The physics level defines mechanical models of components, structures composed of components, and materials
- Multi-physics body model with mass nodes and springs connecting mass nodes:

```
physics part class P = {
  massi: mass object {m:number,..},
  massj: mass object {m:number,..},
  // Use specific spring objects
 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, spting 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,
  ..
}
```

5. Case Study: Multi-domain Simulation

A use-case study combining physical systems and distributed computing used to simulate smart adaptive materials

5.1. SEJAM2

SEJAM: Simulation Environment for the JavaScript Agent Machine

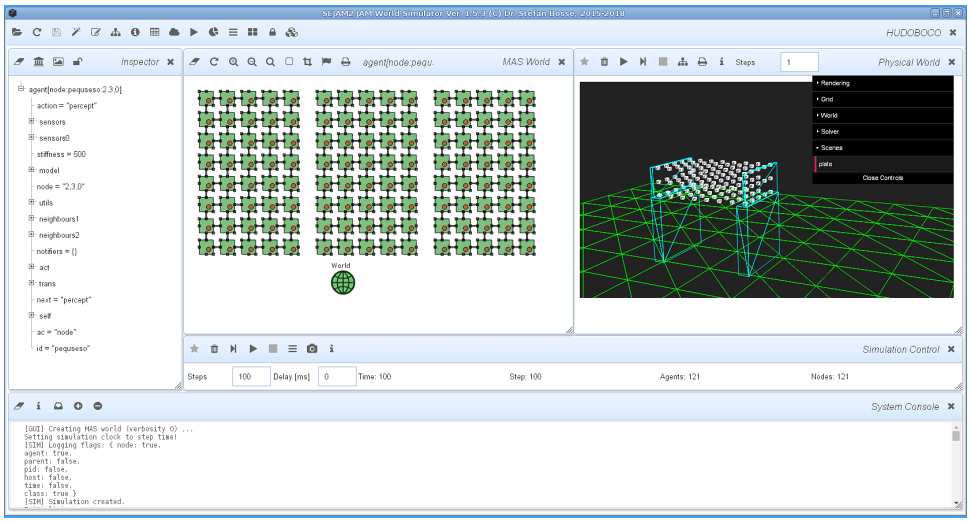
- Coupled physical and computational simulation

Physics
Multi-body Physics Solver

Computation
Multi-Agent Systems and Networks of JavaScript Agent Machines

- Fully JavaScript based modelling and programming
- 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

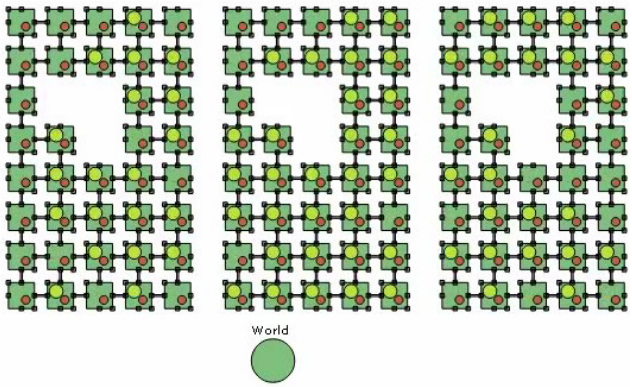
Simulation Front-end and GUI



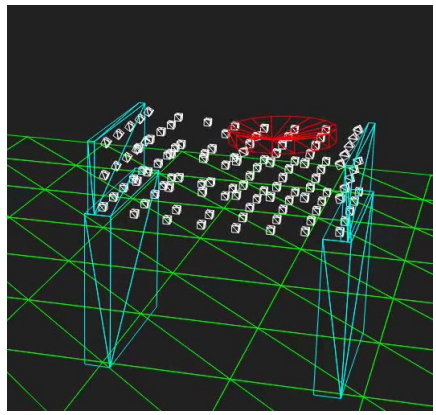
Example: Simulation of Smart Adaptive Materials

MAS World

- Event-based agent behaviour activates sensor processing, distribution, and adaptation only if something changes



Physical World



Simulation Model: JS with Structural name-semantic convention

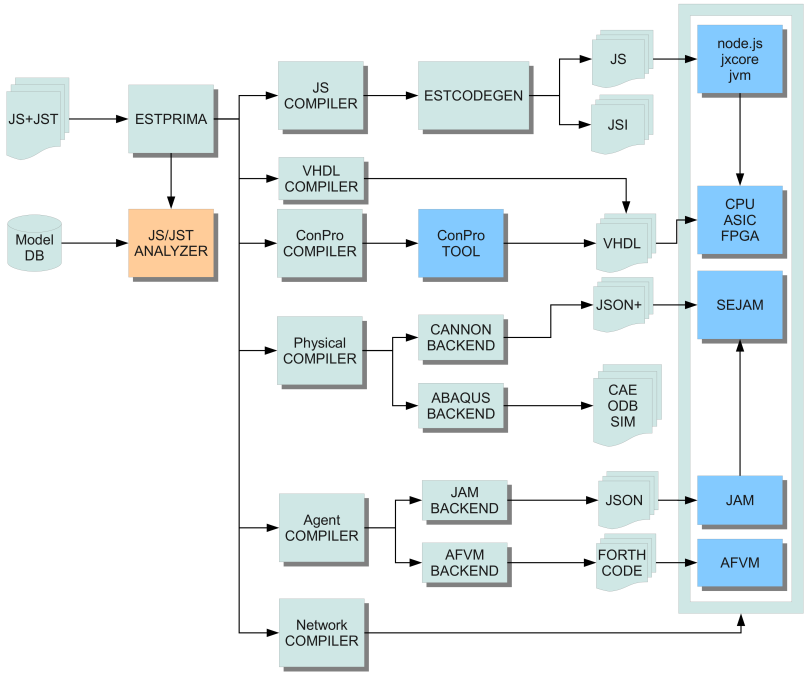
```
simulation : model = {
  // Physical MBP model
  physics : { plate : {..}},
  // Computation and Communication: Agent behaviour Classes
  classes: {
    node   : { behaviour: function () {..}, visiual:{..} },
    broker : { behaviour: function () {..}, visiual:{..} },
    notify : { behaviour: function () {..}, 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
```

Simulation Model: JST type signature overlay for simulation 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 : visual object
}
```

5.2. Model → Simulation → Implementation

- Finally, the models can be used to design and implement the entire system, e.g., a smart sensorial and/or adaptive material with **one unified and unique synthesis framework**



6. Conclusions

- ➤ One unified modelling and implementation (programming) language using JavaScript and a Semantic Type System extensions simplifies the design and implementation of complex perceptive and Cyber Physical Systems
- ➤ All modelling and implementation domains can be covered:

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

- ➤ A use-case demonstrated the benefit of a unified modelling and programming langauge for a multi-domain simulation framework

- ❏ combining physical with computational and communicative simulation
- ❏ using multi-body physics and multi-agent systems.

7. References

1. F. Slomka, S. Kollmann, S. Moser, and K. Kempf, "A Multidisciplinary Design Methodology for Cyber-physical Systems," 2011.
2. M. A. McEvoy and N. Correll, "Thermoplastic variable stiffness composites with embedded, networked sensing, actuation, and control," Journal of Composite Materials, vol. 49, no. 15, 2015.
3. A. Hammad, H. Mountassir, and S. Chouali, "An Approach Combining SysML and Modelica for Modelling and Validate Wireless Sensor Networks," in SESoS 2013 Montpellier, France, 2013.
4. A. Rrustemi, "Simulating Sensor Networks with SystemC," in 14th ESCUGM, Darmstadt, 2006.
5. C. Ernesto Gomez Cardenas, "Modeling Embedded Systems Using SysML," Universidad de Los Andes, 2009.
6. A. U. KHAN, Y. TAKEUCHI, and M. IMAI, "A Sensor Modeling Technique Using SystemC-AMS for Fast Simulation of System-in-Package based Bio-Medical Systems," in SASIMI 2013.