Modellierung und Simulation komplexer Systeme mit annotiertem JavaScript

Stefan Bosse, Universität Bremen, Fachbereich Mathematik und Informatik
Universität Koblenz-Landau, Fakultät Informatik

Der Entwurf und die Simulation komplexer mechatronischer und verteilter intelligenter Systeme erfordern eine einheitliche Systemmodellierungs- und Programmiersprache. Dieser Beitrag stellt JavaScript als eine vereinheitlichte Modellierungs- und Programmiersprache vor, indem JavaScript mit einem semantischen Typsystem JST erweitert wird, um die Lücke zwischen Modellen und Implementierungen zu schließen. Daraus resultiert die JS+ SupersetSprache, die Typisierung, Modellierung und Programmierung kombiniert. Es werden verschiedene Modelldomänen und ihre Beziehung zum JS+Programmierungsmodell einschließlich einiger generischer Transformationsregeln am Beispiel eines sensorischen Materials gezeigt. Schließlich wird das MultidomainSimulationswerkzeug SEJAM eingeführt, das physikalische und datenverarbeitende Simulation mit Agenten kombiniert.

Modeling and Simulation of Complex Systems with Annotated JavaScript

Designing and simulating complex mechatronic and distributed intelligent systems requires a unified system modeling and programming language. This paper presents JavaScript as a unified modeling and programming language by extending JavaScript with a semantic type system JST to bridge the gap between models and implementations. The result is the JS+ Superset language, which combines typing, modeling and programming. Different model domains and their relation to the JS+ programming model including some generic transformation rules are shown using the example of a sensory material. Finally, the multidomain simulation tool SEJAM is introduced, which combines physical and data processing simulation with agents.

Keywords:
simulation, system modelling, sensor materials

Der Entwurf und die Modellierung komplexer und heterogener verteilter Sensor- und Aktuatorsysteme ist eine Herausforderung! Beispiele für solche Systeme sind

- Verteiltes Structural Health Monitoring (SHM) und
- Cyber-Physikalische Systeme (CPS) im Kontext industrieller Produktions- und Fertigungsumgebungen.

Die Modellierung erfolgt dabei auf verschiedenen Abstraktions- und Funktionsebenen:

- System-System;
- System;
- Eingebettetes System;
- Betriebssystem;
- Vernetzung und Kommunikation;
- Verteiltes Rechnen (z. B. unter Verwendung von agentenbasierten Modellen);
- Sensor und Elektronik;
- Hardware; und
- Software mit verschiedenen Modellierungs- und Programmiersprachen.

Die zentrale Frage lautet, wie man solche komplexen CyberPhysikalischen Systeme mit einem einheitlichen Ansatz und Sprache modellieren, entwerfen, simulieren und implementieren kann?

Eine Modellierungssprache drückt Struktur und Beziehung durch einen konsistenten Satz von Regeln aus (Bild 1), die für die Interpretation der Bedeutung von Zusammenhängen verwendet werden. Eine Programmiersprache dient dazu eine konkrete Implementierung eines Modells oder eines Algorithmus durch eine Menge von Anweisungen umzusetzen, die Ausgaben verschiedener Art erzeugen.

Es gibt bereits zahlreiche Modellierungssprachen (M), Simulationssprachen (S), und Programmiersprachen (Implementierung, I) sowie dazugehörige Werkzeuge:

- VHDL → HardwareVerhaltensmodellierungssprachen für digitale Hardware (M, I, S) [11]
- VHDLAMS → Hardwareverhalten Modellierungssprache für digitale und analoge Elektronik (M, I) [1]
- SystemC → Hardware- und Software-Modellierung + Programmiersprache für digitale Hardware; Algorithmische Ebene adressiert eingebettete Systeme (M, I) [2], Simulation (S) [3]
- SystemC-AMS → Hardware- und Software-Modellierung + Programmiersprache für digitale und analoge Systeme (M, I, S) [4]
- JAVA → I.A. nur objektorientierte (OO) Softwareprogrammierung (I)

Dr. Stefan Bosse ist Privatdozent an der Universität Bremen im Fachbereich Mathematik und Informatik sowie Vertretungsprofessor an der Universität Koblenz-Landau im Institut für Softwaretechnik.

sbosse@uni-bremen.de
www.uni-bremen.de

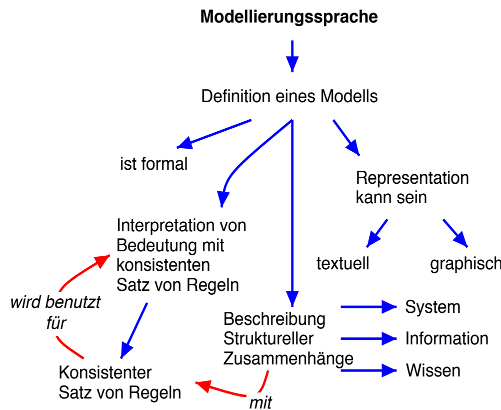

Bild 1: Unterschied zwischen Modellierungs- und Programmiersprachen.

- C++ → Prozedurale und objektorientierte Softwareprogrammierung (I)
- SysML → Modellierung (M) von Software, Hardware und eingebetteten Systemen [5, 6] (M,S)
- Modelica → Objektorientierte, deklarative Multidomain Modellierungssprache für komponentenorientierte Modellierung komplexer Systeme [7] (M,S,I)

Modellierungs- und Programmiersprachen können eine grafische oder textuelle Darstellung besitzen. Übliche Textdarstellungen findet man bei SystemC [2], verwendet für das HardwareSoftwareCoDesign, bei VHDL und Verilog zum Modellieren und Entwickeln von Hardwarekomponenten (Digitale Logik und SoCDesigns, z. B. Implementierung von Sensorknoten, Kommunikationsgeräten usw.); bei C/C++/JAVA für die SoftwareProgrammierung, und bei Modelica, Simulink und Matlab, die in großem Umfang für die Simulation und Modellierung von physikalischen, elektrischen, numerischen und rechnergestützten Systemen eingesetzt werden. Vorhandene Systemmodellierungssprachen, z. B. SysML, einschließlich softwarebezogener UML (nur OO Softwarentwurf), können verwendet werden, um unterschiedliche Bereiche (Domänen) und Ebenen im Entwurfsprozess komplexer Systeme zu adressieren (Bild 2). SysML ist keine eigentliche Textsprache und kann nicht für die Verhaltensimplementierung verwendet werden, sondern ist beschränkt auf die Spezifikation und die strukturelle Beschreibung. Modelica ist eine klassenbasierte, textuelle Modellierungssprache. Es gibt bereits Ansätze,SysML und Modelica zu kombinieren.

In diesem Beitrag wird JST als semantische Erweiterung der weit verbreiteten und etablierten Programmiersprache JavaScript (JS) vorgestellt, die als universelle Modellierungs- und Programmiersprache für den vereinheitlichten Entwurf komplexer verteilter Systeme auf allen Ebenen und Aspekten der Systemspezifikation und -implementierung dient. D. h. JST erweitert JS zu einer MultidomainModellierungssprache. Dadurch wird eine ganzheitliche Modellierung, Simulation, und Implementierung von komplexen Systemen wie Sensornetzwerke auch für nicht auf allen Gebieten fachkundige Entwickler möglich.

Das gesamte Systemmodell, das durch das semantische Typsystem repräsentiert wird und das zum Entwurf komplexer Systeme verwendet wird, besteht aus mehreren verschiedenen Domänen:

1. Physikalische Systeme (Sensoren, Mechanik, ..)
2. Hardware (SoC, analoge und digitale Elektronik, Sensoren, ..)
3. Software und Abstrakte Datentypen (z. B. Graphen)
4. Interaktionsebene (Kommunikation und Protokolle) und Netzwerke (Kommunikationsgraphen)
5. Eingebettete Systeme (Hardware, Software)
6. System und System-of-System
7. Simulation (Multidomain)

Jede Domäne besteht aus mehreren Elementen (Komponenten), die Typen im semantischen System repräsentieren (Bild 3, Darstellung unten). Das Systemmodell kann bezüglich der Domänen und der Modellelemente innerhalb von Domänen stets erweitert werden.

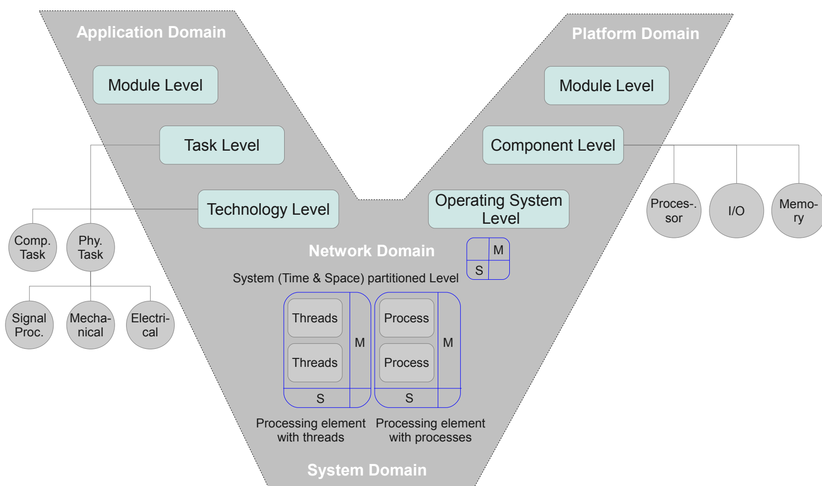

Bild 2: Verschiedene Abstraktionsebenen im Entwurfsprozess und vier Entwurfsdomainen: System, Netzwerk, Applikation, und Plattform (basierend auf [8]).

Das Konzept

Die zentralen Konzepte des JavaScriptProgrammiermodells sind generische Funktionen und generische Objekte. Das JavaScript Semantic Typsystem (JST) fügt Funktionen und Objekten „semantische Annotationen" hinzu und führt Typbeschränkungen und Typsignaturen ein. Eine Typsignatur beschreibt z. B. die Typschnittstelle einer Funktion (Parametertypen und Rückgabetyp). Ursprünglich wurde

JST nur zur Typisierung von Programmen verwendet und wurde getrennt von der Implementierung angegeben.

Der Hauptvorteil von JS gegenüber beispielsweise SystemC ist das Merkmal, dass JS für die Modellierung und die Implementierung von Synthesewerkzeugen verwendet werden kann. Zum Beispiel ist der weit verbreitete JSParser esprima in JS geschrieben und kann leicht erweitert und angepasst werden. Die Verwendung von JS ermöglicht die schnelle Entwicklung von Softwarewerkzeugen. SystemC ist grundsätzlich eine C++ Bibliothek, die es Entwicklern ermöglicht, ein System unter Verwendung der Bibliotheksstrukturen und eines beliebigen C++ Konstrukts (das vom verwendeten Compiler unterstützt wird) sowohl zu implementieren als auch zu simulieren [8].

Im Gegensatz dazu hat JS keine starken Bindungen zu Compilern oder Ausführungsplattformen (JSEngines). JS erfordert kein Programm oder Modulrahmen (d. h. wie eine Bibliotheksumgebung in C++/SystemC). Jedes JSCodeSnippet kann eigenständig ausgeführt werden. Daten und Code können durch das JSON+ Dateiformat (JSON erweitert mit Funktionscode) eindeutig behandelt und ausgetauscht werden. JS beruht auf drei grundlegenden Programmierklassen: Funktional; Objektorientiert; Prozedural. Darüber hinaus ist JS eine dynamisch typisierte Sprache mit einem Programmkontext, der es erlaubt, Objekte zur Laufzeit zu erweitern, um eine Überspezifikation auf Modellebene zu verhindern. Der Vorteil der Verwendung einer weit verbreiteten Sprache anstelle von mehreren verschiedenen Expertensprachen liegt darin, dass die Systemmodellierung und -implementierung von einer größeren Gemeinschaft von NichtExperten und Anwendungsingenieuren durchgeführt werden kann. Dies geschieht durch die Einbettung des vorgeschlagenen Ansatzes in einen ToolboxRahmen, sodass der einheitliche Entwurf von komplexen verteilten Sensor- und CyberPhysikalischen Systemen ermöglicht wird.

Ein Beispiel für ein solch komplexes CPS System ist ein robotisches Material, welches prototypisch in Bild 4 gezeigt ist. Es handelt sich um ein verteiltes Netzwerk aus SensorAktuatorknoten, das für die perzeptive Formänderung einer Struktur (hier ein Balken) genutzt wird [9]. Als Aktuatoren kommen thermoplastische Kunststoffe zum Einsatz, als Sensoren Dehnungsmessstreifen.

Darüber hinaus vereinfacht diese Sprachvereinheitlichung den Einsatz von HighLevelSyntheseansätzen. JS eignet sich aufgrund seines generischen und portablen Programmiermodells, das auf jedem Gerät ausgeführt werden kann, gut für diesen Ansatz.

JS in seiner aktuellen Form kann nicht direkt für diesen Zweck verwendet werden. Um JS an die Anforderungen einer Systemmodellierungssprache anzupassen, wird das Semantische Typesystem (JST) eingeführt, das JS um formale Spezifikationsfunktionen erweitert. JST ermöglicht die Transformation gängiger JS-Sprachkonstrukte (Objekte, Arrays, Funktionen, Variablen) in verschiedene Modellierungstypen (Paket, generische Komponente, Strukturdefinition, Hardwarekomponente, Programm, Berechnungsprozess, Agentenverhalten, numerische Funktion, parametrische Randbedingungen, Anforderungen, Kommunikationsprotokoll, Interaktion, Strukturbeschreibung und vieles mehr). Schließlich konvertiert JST die ursprünglich dynamisch typisierte Programmiersprache JS in eine Programmiersprache mit eingeschränktem Typraum (Typannotation) als eine Voraussetzung für die konforme Programmierung, die spezifizierte Modelleigenschaften zur Laufzeit erfüllt und Mehrdeutigkeiten vermeidet. Semantische Aspekte der Programmierung wie Summentypen, die nicht von JS (und den meisten modernen Programmiersprachen) unterstützt werden, werden von JST

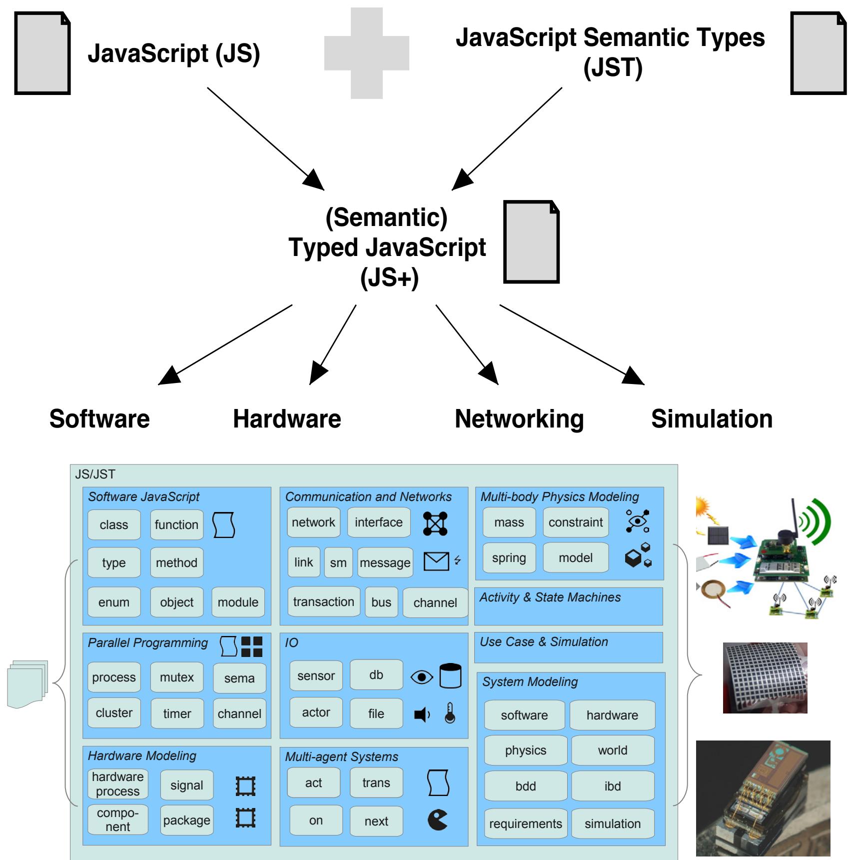

Bild 3: Konzeptioneller Überblick: (Oben) Ein einheitlicher Modellierungs- und Programmieransatz; (Unten) Abbildung der verschiedenen Modellierungsebenen.

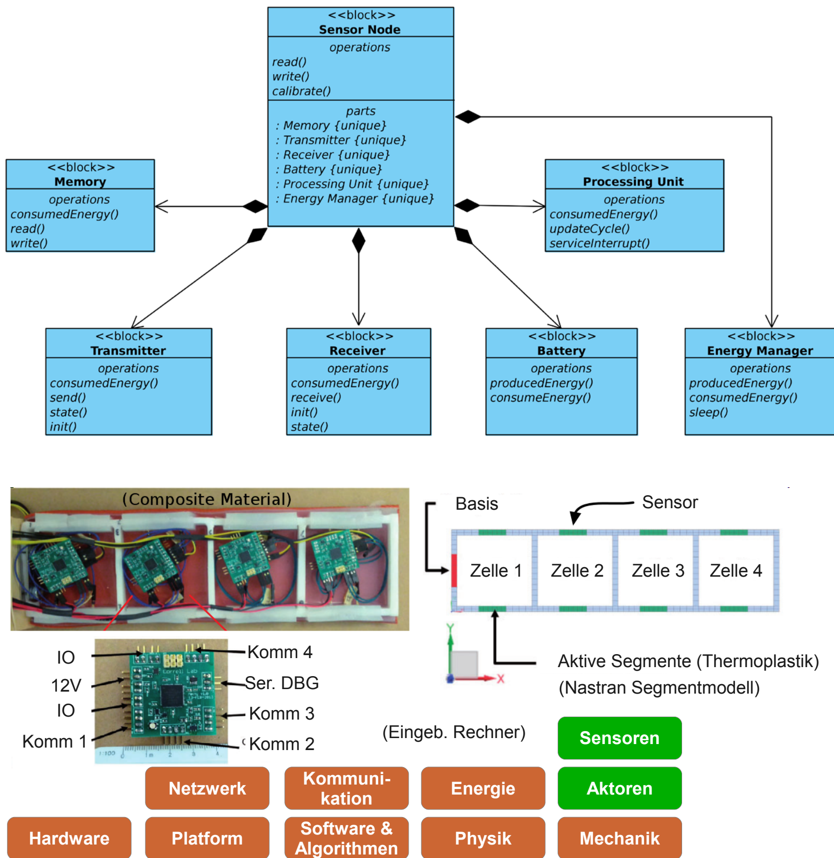

Bild 4: (Oben) Beispiel einer SysMLModellierung eines Sensorknotens; (Unten) Prototyp eines komplexen robotischen Materials bestehend aus einem Netzwerk von Sensor/Aktuator-Knoten [9].

hinzugefügt und erzeugen schließlich einen SuperSet JS+ von JS, wie in Bild 3 gezeigt ist.

Das Grundkonzept der semantischen Erweiterung von JST sind Typkonstruktoren. Die Menge der Basisdatentypen von JS besteht nur aus *number, string, boolean, object, array* und *function*. Durch Typerweiterung mittels Typklassen und Typattributtupeln in der notationellen Form $t_1t_2t_3$ .. t_n wird der Grundtyp t_n durch die zusätzlichen Typannotationen (Typattribute) t_1 .. t_{n-1} semantisch spezialisiert und gebunden. Dabei ist t_{i-1} jeweils ein Untertyp von t_i. Ein Beispiel wäre die Erweiterung des Typs *object* mit der Typklasse *sensor*, Spezialisierung von programmatischen Datentypen wie Arrays durch Hinzufügen des Elementtyps (*number array*), oder Datentypen von Attributen von Objekten mit den Typklassen tag, private, *parameter*. Funktionen in JS können rein prozeduralen, methodischen oder objektkonstruktiven Charakter besitzen, welcher durch die zusätzlichen Typattribute *method* und *constructor* ausgedrückt werden kann. In JST wird Funktionen, Variablen, Objekten und Datenstrukturen ein semantischer Datentyp über eine Typassoziation zugeordnet, die wiederum durch Typdefinitionen komponiert sein kann (siehe Formel 1).

Dabei werden rein prozedurale Datenstrukturen durch das Typsymbol { } bezeichnet, methodenbasierte Objekte hingegen mit dem *object* Typ (wobei Klassen ebenso die { } Notation verwenden, jedoch klar durch den Klassentyp *class* gekennzeichnet). Typdefinitionen können Muster darstellen in denen z. B. Attribute von Objekten mit einem Platzhalter *$id* exemplarisch typisiert sind. Funktionsparameter können durch Angabe des Typs, eines (symbolischen) Parameternamens *@id* oder durch ein NameTypTupel beschrieben werden. Ein Rückgabetyp einer Funktion kann mit einem symbolischen Label versehen werden in der Notation *label:typ*. Obwohl in JS Objekte, Datenstrukturen, und Arrays isomorph sind und alle durch Hashtabellen repräsentiert werden, werden in JST Arrays mit dem Typ *array* bezeichnet. Klassentypen werden wie folgt definiert (und können im Falle von JS+ auch Implementierung enthalten):

```
class C (p1:t1,p2:t2,..) = {
  a : t,
  m : method(p1:t1,p2:t2,..) -> rt
  ..
}
constructor c (p1:t1,p2:t2,..) -> C object
```

Die Menge aller Typklassen ist stets erweiterbar. Jede Typklasse (und auch Typannotation) bindet Semantik an programmatische Konstrukte wie Variablen, Funktionen, und Objekte. Jede zusätzliche Typerweiterung bindet stärkere und genauere Semantik.

Es gibt zwei Ansätze JS mit semantischer Beschreibung zu verbinden:

- Verwendung von unverändertem JS mit einer getrennten Typüberlagerung (JST Overlay), d. h. es wird zunächst eine Namenssemantik verwendet, die getrennt mit Typsemantik verknüpft wird.
- Verwendung einer modifizierten JS+Notation, welche die semantische Typannotation in JS integriert, wobei Ansatz (1) inbegriffen ist.

Das folgende (unvollständige) Beispiel eines Modells (inklusive APIImplementierung) eines aktiven Sensorknotens mit einem Temperatursensor soll Ansatz (1) verdeutlichen, im Vergleich zu dem SysMLModell aus Bild 4 (Formel 2).

Formel 1

JST

```
type t =  expression  (t1,t2,t3,..)
typeof x = t
function f(p1:t1,@p2,..) -> rt

x : t1         Kurzform
o : { a : number, b:   array, .. }
r : t2
type t2 = { $entry : number array }
```

JS+

```
type t =  expression  (t1,t2,t3,..)
typeof x = t
function f(p1:t1,@p2,..) -> rt {
  implementation   }
var x : t
var o = { a:number = 1, b:array = [1,2,3] }
r = { p1=[1,  -1,0], p2=[0,1,1], p3=[1,1,1] }
type t2 = { $entry : number array }
```

```
JS
function tempS (init) {
 this.kind="temperature"
  this.val=init
 this.k=1
  this.cputime=0
}
temp.prototype.calibrate = function (x,y) {
 this.k=y/x
}
temp.prototype.read = function () {
  return this.transfer(this.val)
}
temp.prototype.write = function (x) {
 return this.val=x
}
temp.prototype.transfer = function (x) {
  return this.val*this.k
}
temp.prototype.consumedEnergy = function () {
 return this.cputime*ECPUK
}

function tempSN (params) {
 this.sensor = new tempS(params.init)
 this.battery = new batteryS(params.charge)
 this.memory = new memoryS(params.bytes)
 this.receiver = new receiverS(params.freq1)
 this.transmitter = new senderS(params.freq2)
 this.processor = new cpuS(params.cpufreq,..)
 this.manager = new emS(params.idle,..)
}
```

```
JST
sensor class TempS (init:number) = {
 kind: tag string,
 val : private data number,
 k: parameter number,
 cputime: model number,

 calibrate: api method (x:number,k:number),

 read: api output method ()  -> number

 write: api input method (number)

 transfer:   physical model method (input)   ->
 output

 consumedEnergy   : model method ()   ->
 model  number

}
constructor tempS() -> TempS

batter class BatteryS;
memory class MemoryS; ..

node sensor class tempSN (params) = {
 sensor: sensor class TempS,
 battery : battery class BatteryS,
 memory : memory class MemoryS,
 receiver : receiver class ReceiverS,
 transmitter : sender class TransmitterS,
 processor : model class CpuS,
 manager : manager class EmS
```

Formel 2

```
JST
typeagent constructor =
 function (parameter {}) > agent class object
type agent class = {
  $identifier : body attribute, ..,
  act :         activitiy {} ,
  trans :       transition {},
  on:           signal handler {},
  next:         $activity | string,
}
type activity function = computational blockable function ()
type activity {} = { $actname  : activity function (), .. }
type transition function = computational function ()-> activity string
type transition {} = { $activity : $activity | transition function, .. }
typeof $activity = identifier | string
type signal handler function = computational function (arg1,arg2,..)
type signal handler {} = { $signame : signal handler function, .. }
type parameter = number|boolean|string|array|record
type record = {}

agent constructor $agent-class-name (parameter {})-> agent class object

JS function agent-class-name (par1,..) {
Body Variables and private aux. Functions
 this.xi= 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 = { signali : function (arg,from) { .. }, .. }
 this.next = ai;
}
```

Formel 3

Diese semantische Typannotation ist informativer und präziser als das in Bild 4 gezeigte SysMLDiagramm, welches keine dezidierte Unterscheidung zwischen Simulation, Modellierung und Implementierung machen kann.

In JS+Notation werden schließlich Implementierung (JS) und Semantische Typannotation (JST) vereint:

```
JS+
sensor sensor class TempS (init:number) = {
 kind: tag string = "temperature",
 val private data number = init,
 k: parameter number = 1,
 calibrate: api method (x:number,k:number) {
  this.k=y/x
 },
 read : api output method ()-> number {
  return this.transfer(this.val)
  }, ..}
```

MultidomainModellierung und Simulation

Im folgenden Abschnitt sollein Anwendungsbeispiel für die semantische Typannotation in der Modellierung, Implementierung und Simulation komplexer mechatronischer Systeme anhand des SEJAMSimulators und eines robotischen Materials gezeigt werden. Einzelheiten finden sich in [10].

Multiagentensysteme

Diese Domain bildet die Modellierung und Implementierung der Datenverarbeitung und digitaler Kommunikation in Sensor- und Aktuatornetzwerken durch Agenten ab (siehe Formel 3).

Mechanische Multikörpersysteme

Diese Domain bildet die Modellierung von physikalischen Körpern und Strukturen mittels eines Multikörperphysikmodells ab (Multibody Physics, MBP). Das Modell besteht aus einem Netzwerk aus Masseknoten, die über (parametrisierbare) Federn in einer regulären Struktur miteinander verbunden sind. Nachfolgend ist die Typsemantik des physikalischen Systems beschrieben (siehe Formel 4).

MultidomainSimulationsmodell

Im folgenden Abschnitt ist ein JS/JSTBeispiel der Modellierung eines robotischen Materials gezeigt (ähnlich dem in Bild 4), welches für die MultidomainSimulation verwendet wird und aus einem Datenverarbeitungsteil mit einem Multiagentensystem und einem physikalischen Teil besteht. Das Multiagentensystem wird in einem Netzwerk aus Plattformen ausgeführt, und das physikalische Modell wird

```
JST
physics type element = mass {} | spring {} | sensor {}
physics model class MBP = {
 mass: mass constructor function ( m:kg,pos:position,..)-> mass,
 spring: spring constructor function (k:number,d:number, l0:number,..)-> spring,
 constri: constraint function (x,y,..) -> z,
 connect : function (n1:mass,n2:mass)  -> spring|null,
 constructor : function (size_n,size_m,..)-> network graph,
  ..
}
type spring = model object { force: method (n1:mass,n2:mass)  -> force:number , .. }
type mass = graph node
type kg = physical number
type position = logical [x:number,y:number,z:number]
```

Formel 4

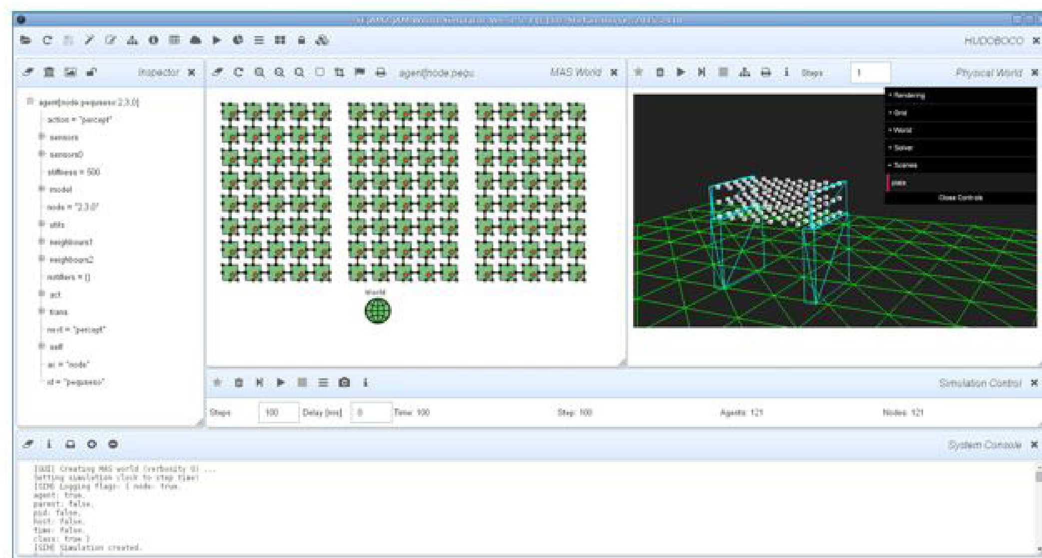

Bild 5: Der Multidomain-Simulator SEJAM2 kombiniert physikalische Mehrkörpersimulation (rechts) mit der Simulation (und Ausführung) von Multiagentensystemen und Plattformnetzwerken (links).

durch ein Multikörpersystem gebildet. Der Simulator deckt die folgenden Modellierungs- und Implementierungsdomänen ab:

- Mechanische Struktur auf physikalischer Ebene
- Prozess (Datenverarbeitungsmodell, JavaScript)
- Mobile Agenten (Verteiltes Datenverarbeitungs- und Kommunikationsmodell)
- Agentenplattform (JAM)
- ICT Netzwerke, Kommunikationsprotokolle
- Visualisierung und Nutzerschnittstellen (HMI)

Auch hier wird wieder der Ansatz einer Namenssemantik mit einer getrennten Typüberlagerung genutzt. So bedeutet das Simulationsattributphysics die Modellierung des physikalischen Systems, classes modelliert das Verhalten und die Visualisierung von Agenten (siehe Formel 5).

Dieses parametrisierbare Simulationsmodell, vollständig in JS/JSON+ formuliert, wird dann für die Simulation und Evaluierung eines künstlichen robotischen Materials verwendet, in dem selbstorganisierende Agentensysteme die Struktur durch Topologieoptimierung (Anpassung von Aktuatorsteifigkeiten) an verschiedene Last- und Schadenssituationen anpasst. Eine typische Simulationssituation mit dem ICTNetzwerk und Agenten auf der einen Seite und dem MBPModell der Struktur auf der anderen Seite, ist in Bild 5 gezeigt.

Zusammenfassung

Eine einheitliche Modellierungs-, Simulations- und Implementierungssprache (Programmierung), die JavaScript mit semantischer Typerweiterung verwendet, vereinfacht den Entwurf und die Implementierung von komplexen perzeptiven und CyberPhysikalischen Systemen. Alle Modellierungs- und Implementierungsdomänen können abgedeckt werden, Modellierung und Implementierung können kombiniert werden. Ein Anwendungsfall zeigte den Vorteil einer einheitlichen Modellierungs- und Programmiersprache für ein MultidomänenSimulationsframework mit einer Vereinigung von physikalischer mit rechnerischer und kommunikativer Simulation unter Verwendung von Multikörpern, Physik und Multiagentensystemen.

JS

```
var simuPart = {
 Physical MBP model
 physics : {
  plate : {
     mass : function (m,pos,..) { .. },
   spring : function (k,d,l0,..) { .. },
     constructor : function (parameter) { .. }, ..
  }
 },
 Computation and Communication: Agent behaviour Classes
 classes: {
  node   : { behaviour: function () {..}, visual:{..} },
  broker : { behaviour: function () {..}, visual:{..} },
  notify : { behaviour: function () {..}, visual:{..} },
  world  : { behaviour: function () {..}, visual:{..} },
 },
 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 setup and initialization
 world: { ..
    init : { agents: {..}, physics: {..}},
    meshgrid : { node:{..}, port = {..},
         link:{..} .. } } // Network model
```

JST

```
simuPart : simulation model
```

Formel 5

Schlüsselwörter:
Simulation, Systemmodellierung, sensorische Materialien

Literatur

[1] Chapuis, Y.-A; Zhou, L.; Fujita, H.; HervÃ, Y.: Multi-domain simulation using VHDL-AMS for distributed MEMS in functional environment: Case of a 2D air-jet micromanipulator. Sensors and Actuators A: Physical (2008) 148, S. 224238.
[2] Stoppe. J.;Drechsler, R.: Analyzing SystemC Designs: SystemC Analysis Approaches for Varying Applications, Sensors MDPI (2015), S. 1039910421.
[3] Rrustemi, A.: Simulating Sensor Networks with SystemC, in 14th ESCUGM. Darmstadt 2006.
[4] Farooq, M.; Adhikari, S.; Haase, J.; Grimm, C.: Modeling methodology in SystemC-AMS for embedded analog mixed signal systems, in Proceedings of the 8th International Conference on Frontiers of Information Technology - FIT '10. New York 2010.
[5]Cardenas, C. E. G.: Modeling Embedded Systems Using SysML. Universidad de Los Andes, Columbia 2009.
[6] Maissa, Y. B.;Mouline, S.: A SysML profile for wireless sensor networks modeling, in I/V Communications and Mobile Network (ISVC), 5th International Symposium on.Rabat, Morocco 2010.
[7] Hammad, A.; Mountassir, H.; Chouali, S.: An Approach Combining SysML and Modelica for Modelling and Validate Wireless Sensor Networks, in SESoS 2013. Montpellier, France 2013.
[8] Slomka, F; Kollmann, S.; Moser, S; Kempf, K.: A Multidisciplinary Design Methodology for Cyber-physical Systems.Ulm 2011.
[9] McEvoy, M. A.; Correll, N.: Materials science. Materials that couple sensing, actuation, computation, and communication, Science. University of Colorado at Boulder, USA 2015.
[10] Bosse, S.; Lehmhus,D.: Adaptive Materialien mit Multiagentensystemen, Industrie 4.0 Management 4 (2018), ISSN 23649208.
[11] P. J. Ashenden, The Designer's Guide to VHDL. Morgan Kaufmann, 2008