Open Access

Open Journal of Web Technology (OJWT)
Volume 8, Issue 1, 2021

www.ronpub.com/ojwt
ISSN 2199-188X

From JSON to JSEN through Virtual Languages

Antonello Ceravola, Frank Joublin

Honda Research Institute Europe GmbH, Carl Legien Str. 30, Offenbach/Main, Germany,
{Antonello.Ceravola, Frank.Joublin}@honda-ri.de

ABSTRACT

In this paper we describe a data format suitable for storing and manipulating executable language statements that can be used for exchanging/storing programs, executing them concurrently and extending homoiconicity of the hosting language. We call it JSEN, JavaScript Executable Notation, which represents the counterpart of JSON, JavaScript Object Notation. JSON and JSEN complement each other. The former is a data format for storing and representing objects and data, while the latter has been created for exchanging/storing/executing and manipulating statements of programs. The two formats, JSON and JSEN, share some common properties, reviewed in this paper with a more extensive analysis on what the JSEN data format can provide. JSEN extends homoiconicity of the hosting language (in our case JavaScript), giving the possibility to manipulate programs in a finer grain manner than what is currently possible. This property makes definition of virtual languages (or DSL) simple and straightforward. Moreover, JSEN provides a base for implementing a type of concurrent multitasking for a single-threaded language like JavaScript.

TYPE OF PAPER AND KEYWORDS

Regular Research Paper: *JSEN, JSON, executable data structures, homoiconic languages, metaprogramming, virtual languages, concurrency, threading*

1 INTRODUCTION

There is a large variety of data formats available for storing or transferring information between different software systems. Some of them are focused on specific domains while others are general purpose and can be easily used cross-domain. In this second group, XML is one of the most well-known data formats, designed in 1996 by the XML Working Group [7] [36] with the target of being a general-purpose data format, easy to read and usable on Internet applications. XML received a lot of attention from different communities [33] leading to its usage in domains such as government, chemistry, telecommunication, astronomy, and several others. However, along its wide usage, XML has been criticized for its verbosity and complexity [33][35], consequently giving space for other data formats to emerge [34]. One of the most used alternatives is JSON [15][16]. Considered lighter, simpler, and more readable than XML [27][37]; JSON belongs to the family of general-purpose data formats, and continues gaining space in new domains, particularly among Internet applications.

We use JSON data structures for storing and transmitting data across different subsystems in one of our projects, a web-based, client-server avatar system. It is implemented with HTML, CSS and JavaScript, used in both client and server side [32]. For some

specific functionalities we came across the requirement of storing and manipulating code programmatically, in ways not possible with JavaScript. We had to programmatically control function's execution, in some cases we had to execute functions in a concurrent way, as well as injecting or removing code statements on them. This type of code manipulation is supported by homoiconic languages: "*In a homoiconic language, the primary representation of programs is also a data structure in a primitive type of the language itself*" [4][25].

JavaScript provides a degree of homoiconicity [6] allowing introspection of classes, prototypes, and members. It allows dynamic changes like adding/ removing or redefining members in classes/objects. JavaScript gives access to functions, by allowing access to their names, parameters, or their full source code (as a single string by applying the method ".toString()" to a function identifier). However, JavaScript homoiconicity do not go beyond this; for instance, it is not possible to have access to function's individual statements or parts of them. A greater level of introspection/manipulation, instead, can be found in homoiconic languages like Lisp [31], SmallTalk [11] or Tcl [28] and in the JavaScript macro language of Majaho [14], as well as EsLisp [19]. In particular, the latter implements a Lisp based e-expression syntax for JavaScript, where programs are natively stored in data structures fully accessible in a programmatic way by programs themselves.

In this paper, we describe a data structure that allows a finer degree of access to function statements. We called this format JSEN, JavaScript Executable Notation (name inspired by the symmetric relations between JSEN and JSON). Where JSON is a data format for storing and manipulating objects (data), JSEN is a format for storing and manipulating executable code. Furthermore, with JSEN we can easily handle asynchronous functions without falling into the callback-hell pattern. In this paper, we describe the JSEN format, giving a glimpse on some of the interesting features we could derive from that. JSEN is available as Open-Source Software in GitHub at https://github.com/HRI-EU/JSEN. All necessary libraries are present, with examples and all source snippet mentioned in this paper. We will now draw the reasons that led us to introduce JSEN before we give an overview of the structure of the paper.

1.1 MOTIVATION

Our experience with JavaScript has put in light several limitations that JSEN is trying to overcome. The starting point for us has been the needs to programmatically control function's execution, to manage asynchronous code as well as the need to easily implement concurrent processing particularly important in artificial intelligence systems. On the way to the proposed solution, we discovered that object oriented and functional programming provided by JavaScript was not sufficient for the problem we were tackling and therefore we needed language extensions which were not easily possible in JavaScript. For example, our applications had the needs to model event based concurrent programs which, through JSEN became easily implementable.

In the next paragraph we are going to describe the structure of this paper, touching the different concepts that will be elaborated in the different sections.

1.2 OVERVIEW

In this paper we start the introduction of JSEN, in the paragraph 2, by first looking at JSON. The two data formats are similar and complementary. This is an important relation that facilitate the introduction of this work. In the paragraph 2.2, by describing JSEN, we introduce its core principles on which it is based: closure and heterogenous multi-dimensional arrays. At that stage we explain the syntax, we describe how JSEN extends JavaScript homoiconicity, we introduce the concept of JSEN virtual machine and virtual languages. In the paragraph 2.3 we show how to go from the definition of a JSEN data structure to its execution. Looking at JSEN as a storage format, in the paragraph 2.4 we show how, similarly as JSON, JSEN can be used to transfer programs across systems.

We then describe, in the paragraph 3, the architecture of JSEN, to understand how JSEN can be used in applications. The paragraph 3.1, goes deeper into virtual languages, introducing how JSEN data structures can be executed. In the paragraph 3.2 we then investigate more details on the memory representation of JSEN by stepping into the stages of JSEN definition, compilation, and execution. Existing concurrent/asynchronous methods available in JavaScript are then compared with JSEN in the paragraph 3.3. The paragraph 3.4 describes in more details how a JSEN virtual machine works and how it can execute JSEN data structures in a concurrent way.

We complete our analysis in the paragraph 4 by showing the main properties of JSEN like extended homoiconicity for the hosting language, introduction of code serialization, execution performance of JSEN vs native code, virtual language support/extension and execution of concurrent code. We close the paper with the paragraph 5 by shortly describing what has been intentionally left out from this paper, which will then be addressed in follow up papers, and we give a

summary and conclusions in the paragraph 6. Let us start now with the origin of the name.

2 FROM JSON TO JSEN

For a more gradual introduction we show here how JSEN can be easily understood by looking at its relations with JSON.

2.1 JSON

JSON is a human readable data format based on the JavaScript language, shaped around the syntax of JavaScript object literals [18]. It has been defined with the purpose of encapsulating data that could be used or manipulated in a program as well as used as data exchange/storage format. The elements of a JSON data structure are simple types like numbers, strings, Booleans, null values together with complex types like arrays and objects. The inclusion of arrays and objects introduce the possibility to create hierarchies. In Listing 1 we show an example of a JSON data structure describing a business card:

```
{
  “Name”: “James”,
  “Surname”: “Bond”,
  “Position”: “IO - Intelligence Officer”,
  “Company”: “MI6”,
  “Address”: {
    “Street”: “Albert Embankment, Vauxhall”,
    “Number”: 85,
    “PO”: “SE11 5AW”,
    “City”: “London, UK”
  }
  “Telephone”: +007
}
```

Listing 1: Example of JSON data structure (business card)

The usage of object literals, acting as associative arrays, gives no limits to the complexity of the data structures that can be represented in JSON. This data format is used in many different domains, it became an open standard [17] and many different languages [15] and applications support it. One of the major key factors for the success of JSON is its simplicity and its readability.

2.2 JSEN

Similarly, to JSON, JSEN is a simple human readable format based on the JavaScript language, shaped around the concept of JavaScript arrays. It has been defined with the purpose of encapsulating algorithms, programs or functions that could be executed or manipulated in a program or that could be used as exchange/storing format. Elements of JSEN can be anonymous functions, pure JSEN statements, strings, arrays and objects. In Listing 2 we show an example of a JSEN data structure for a program that implements the computation of prime numbers.

The choice of using arrays as supporting structure for JSEN, comes from the needs of representing sequences of statements of a program: arrays elements are ordered and can represent a sequence. Programs are also constituted of blocks and sub-blocks; similarly, arrays in JavaScript can be nested. In this way, it is possible, in JSEN, to express any algorithm with arbitrary nesting and complexity.

Let us look at JSEN from a closer perspective. The example (1) in Listing 3 shows an empty JSEN data structure (empty array): empty program. The example (2) in Listing 3 shows how to specify a comment. In JSEN comments are defined through strings, allowing persistence of them into JSEN data structures.

```
[
  ()=> result = '',
  'Start finding prime numbers from startNumber',
  ()=> number = startNumber,
  'Compute till next 100 numbers',
  JSEN.for( 'i', 1, 100 ),
  [
    ()=> { countDivisors = 0;
           nTest = number; },
    JSEN.while( ()=> nTest <= 1 ), // Check divisors
    [
      JSEN.if( ()=> i%nTest == 0 ),
        ()=> ++countDivisors,
      ()=> --nTest,
    ],
    'Prime number found if it has only 2 divisors',
    JSEN.if( ()=> countDivisors == 2 ),
      ()=> result = result+i+' ',
    ()=> ++number, // Go to next number
    ],
  ()=> console.log( 'Prime numbers found '+result ),
]
```

Listing 2: Example of JSEN program (computation of prime numbers)

The example (3) in Listing 3 shows the encapsulation of a JavaScript statement. In JSEN, this is done through JavaScript anonymous functions, allowing the storage of valid JavaScript statements, which can be evaluated at a later point in time.
Moreover, thanks to JavaScript closures, such statements have access to global or local variables defined in the context around them. This way of defining statements gives JSEN data structures access to the full JavaScript language.

Anonymous functions are used here to get “pointers to statements”. These pointers are stored in a JSEN data structure together with the context on which they are defined. The creation of such “statement-pointers” is done through JavaScript anonymous function

definition *p = ()=>{}*. Dereferencing such "pointers" is done via function call *p()*. JSEN makes an extensive use of anonymous functions and closure.

```
(1)   Empty JSEN
      []

(2)   JSEN Comment
      [ 'this is a comment' ]

(3)   Statement
      [ ()=> console.log( "Hello World" ) ]

(4)   JSEN sub-block
      [
        ()=> console.log( 'First message' ),
        [
          ()=> console.log( 'Second message' ),
        ],
      ]

(5)   JSEN Statement
      [
        JSEN.if( ()=> a > 1 ),
        [
          ()=> console.log( "Condition true' ),
        ],
      ]
```

Listing 3: Basic JSEN data structure

Example (4) in Listing 3 shows how to define blocks and sub-block in JSEN, where the root block of the example has two elements: the first element with *()=> console.log('First message')* and the second element with an array containing the statement *()=> console.log('Second message')*. Thanks to the way JavaScript handles heterogeneous multidimensional arrays, it is possible to have arrays that contain arrays with different sizes and different data types. JSEN comes with a small set of pure JSEN statements giving the possibility to perform a set of basic language control statements and few other operations in a "JSEN-way". We will explore the reasons for that at a later stage. Example (5) in Listing 3 shows the usage of pure JSEN statements: *JSEN.if()*. It should be noted that the argument of *JSEN.if()* is an anonymous function that returns the value of the condition, evaluated only when the *JSEN.if()* statement is executed.

Let us come back to the way JSEN encapsulates JavaScript statements. Listing 4 shows an example of a JavaScript code with a JSEN data structure (*jsenTest*). The first lines (1, 2, 3) contains the definition of JavaScript variables. Line 4 contains the definition of a JSEN data structure encapsulating two statements: *()=> output[0] = input[position]* and *()=> console.log(output)*. Those statements make use of the variables defined before.

```
1  let position = 0;
2  let input = 'aabbb';
3  let output = '';
4  let jsenTest = [
5    ()=> output[0] = input[position],
6    ()=> console.log( output ),
7  ];
8  console.log( output );
9  JSENVM.run( jsenTest );
```

Listing 4: Encapsulation of JavaScript statements into JSEN data structures

This is possible thanks to the fact that such statements are anonymous functions, able to access variables through closure [2]. It is important to note here that a JSEN data structure (i.e. *jsenTest*) contains only functions definition, therefore the statement *console.log()* in line 8, once executed, will print the variable output with the value assigned in line 3 (empty string). Execution of the JSEN data structure is done in line 9, through the call *JSENVM.run(jsenTest)*. This function triggers the execution of the statements in lines 5 and 6, which will print to the console the value of output: 'a'. The *JSENVM.run()* function is a static function provided by JSENVM, the JSEN virtual machine (described in the next paragraphs). This function is available for a handy execution of JSEN data structures.

Execution of JSEN data structures looks very similar to execution of normal functions. Let us now compare the previous JSEN example with an equivalent in pure JavaScript. Here in Listing 5 we can make a direct comparison.

```
1 let position = 0;
2  let input = 'aabbb';
3  let output = '';
4  let jsenTest = [
5    ()=> output[0] = input[position],
6    ()=> console.log( output ),
7  ];
8  console.log( output );
9  JSENVM.run( jsenTest );
```

```
1 let position = 0;
2  let input = 'aabbb';
3  let output = '';
4  function jsenTest() {
5    output[0] = input[position];
6    console.log( output );
7  }
8  console.log( output );
9  jsenTest();
```

Listing 5: Comparison of JSEN example with a pure JavaScript equivalent

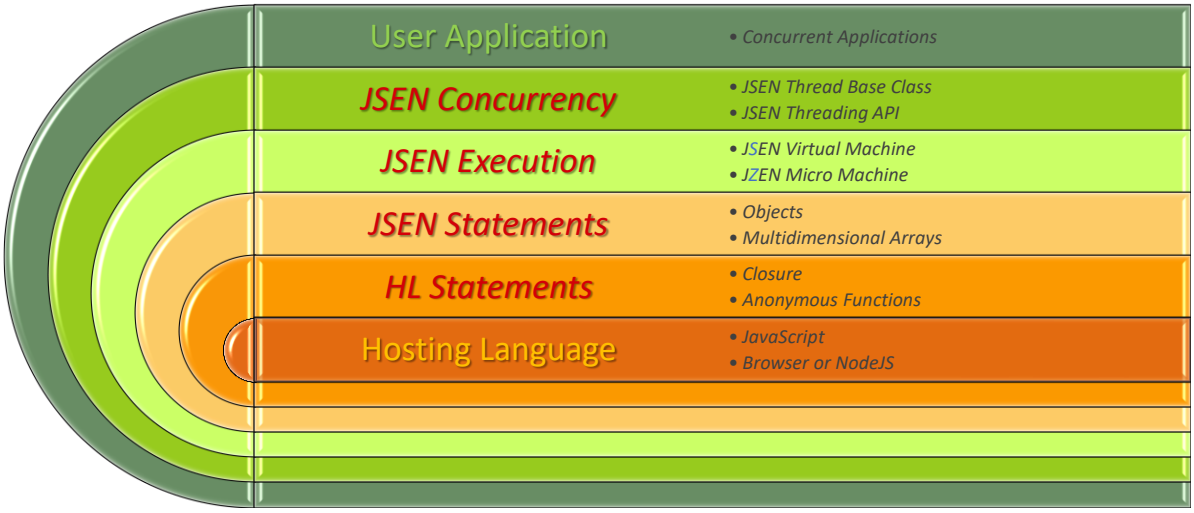

Figure 1: JSEN Architecture

Both examples implement the same functionality, however, there are a few differences to be considered between the JSEN example and the pure JavaScript one. Writing JSEN data structures requires keeping in mind the following notions:

- JSEN statements are stored into arrays.
- JSEN JavaScript statements should always be encapsulated into anonymous functions (in order to be executed only once a JSEN data structure is executed), therefore they should start with '*()=>*' and end with a comma.
- Exception on the previous point is when using pure JSEN statements, which starts with "*JSEN.*". Such statements are in the form of "*JSEN.<statementName>(<parameters>)*". Like the one we have seen in Listing 2: *JSEN.while()*, *JSEN.if()* and *JSEN.else()*. Such statements are necessary to model control flow at the level of JSEN code. This is important for ensuring a deep level of homoiconicity, opening the possibility to operate programmatically on control statement (metaprogramming) and allowing time-sharing execution of different JSEN threads (see the paragraph 3.4).
- Declaration of variables should not be done into JSEN statements (e.g. *()=> let a = 1,*), since such variable would be (according to JavaScript semantic) only visible in the related anonymous function and would not be accessible by other JSEN JavaScript statements. Therefore variables, as shown in our examples, should be declared before the definition of JSEN data structures; Nevertheless, pre-declared variables can be modified into JSEN statements.
- To execute a JSEN data structure, it is possible to use the public static function *JSENVM.run(<jsenDataStructure>)* available in the JSENVM virtual machine (however the API provide other ways to execute JSEN data structures; refer to the Git repository for more information on that).

One important difference between JSEN and pure JavaScript code, in terms of homoiconic capabilities, is the granularity at which function's body can be accessed. In pure JavaScript, the full body is returned as a string using *jsenTest.toString()*. JSEN, instead, gives access to each individual statement (Listing 5). This property contributes to extend the capability of JavaScript metaprogramming [23]. Let us look now at the definition of a JSEN data structure and its execution.

2.3 Definition vs. Execution Time

In the evaluation of a JSEN statement it is important to notice that there is a difference between definition and execution time. Let us take the example in Listing 6 where there is a variable *'a'* declared with the value *0* in line 1. When JavaScript parse the variable *jsenTest* defined in line 2, it will instantiate an array in memory, evaluating each element of the array. We call this phase definition-time. It is important to notice the difference of the evaluation of line 4 and line 6. In both lines there is the execution of the function *JSEN.if()*, in line 4 the parameter is 'a == 0' while in line 6 the parameter is '()=> a == 1'. The former parameter is evaluated at

definition-time therefore it will be evaluated with the value of the variable at line 1 (resulting in the value *'true'*). The latter is evaluated as an anonymous function pointer at definition-time. Therefore the value will be evaluated only when *JSENVM.run()* will execute it. We call this phase execution-time. The parameter of the *JSEN.if()* at line 6 is therefore evaluated with the value of the variable modified at line 3 (resulting in the value *'true'*).

```
1  let a = 0;
2  const jsenTest = [
3    ()=> a = 1,
4    JSEN.if( a == 0 ),
5      JSEN.print( 'a is 0'),
6    JSEN.if( ()=> a == 1 ),
7      JSEN.pring( 'a is 1' ),
8  ];
9  JSENVM.run( jsenTest );
```

Listing 6: Definition vs. execution time

This is an important difference that should be considered every time a parameter is passed to a JSEN statement. To give a parallel with the C languages, the definition-time correspond to the preprocessor, while the execution-time correspond to run-time. Let see now how JSEN data structures can be serialized.

2.4 Serializing JSEN

In JavaScript, an object in memory can be sent to another program through JSON serialization function. *JSON.stringify(object)* transforms an object into a string which can be sent and re-instantiated by using *JSON.parse(jsonString)*. The same can be done with JSEN by using *JSEN.stringify(jsenData)* and *JSEN.parse(jsenString)*.

In the same way as JSON deserialization requires knowledge about the expected received data, JSEN deserialization requires a context of free variables used in the received JSEN program. For instance, in transmitting the JSEN data structure shown in Listing 6, the receiving program should have a variable '*a*' in its receiving context. Let us turn now to the way JSEN is hosted in JavaScript and how a programmer can access it by looking at its architecture.

3 JSEN ARCHITECTURE

We have introduced JSEN in the context of the JavaScript language; however, JSEN is based on a set of computer language principles that can be found in other languages too. JSEN can be easily ported to languages which provides the necessary principles. Let us drill down further and look at the architecture of JSEN described in Figure 1.

At the core of the architecture, in this figure, we find the hosting language (HL), JavaScript in our case. On top of that, JSEN uses hosting language's anonymous functions and closure to define statements (case (3) of Listing 3). JSEN then makes use of objects and multidimensional arrays. Objects are used to represent pure JSEN statements (e.g. *JSEN.if()*) while multidimensional arrays are used to represent blocks/sub-blocks of statements. This is the base for defining JSEN data structures to represent algorithms, programs, or functions. As already mentioned, the execution of JSEN data structures is implemented in JSENVM, a virtual machine that implements the logic for executing JSEN data structures with all its statements. A smaller version of the JSEN virtual machine, named JZENVM, with a minimal set of functionalities, has been created for the implementation of tests for JSENVM API.

The concept of JSEN gives the possibility to implement, through the JSEN virtual machine, a form of concurrent multitasking. The JSENVM provides an API for handling concurrent tasks, together with an additional JSENThreadClass base class, allowing creation of active-objects [20]. By using JSENVM/JZENVM and JSENThreadClass developers can make use of concurrency in their applications in a very simple way. JSEN's concurrency is very similar to the concept of coroutines [24], available to several languages. Let us have a look now on how JSEN gives the possibility to extend the hosting language via virtualization.

3.1 Virtual Language

Virtualization, in the computer science domain, provides a new level of flexibility for handling computers, operating systems, file systems or complete sets of applications. For instance, through the usage of virtual machines it is possible to host, on the same hardware, different machines with different resource configurations, operating systems, and devices, together with an easy way to start/stop and control them, even in a programmatic way. The same can be done with virtual file systems and containers, where a portion of a file system can be added/removed programmatically, giving access to storage, without any persistent change in the hosting file system. In relation to the concept of virtual languages proposed by [3], JSEN act as a platform for virtual languages for the host language on which it runs (JavaScript in our case). JSEN introduces a set of new language statements that give additional control mechanisms for programs, extending but not interfering with the hosting language. The JSEN statements shown in Listing 7 are some of the ones we currently defined.

Control Statements
- JSEN.if
- JSEN.else
- JSEN.loop
- JSEN.while
- JSEN.for
- JSEN.foreach
- JSEN.until
- JSEN.switch
- JSEN.case
- JSEN.label
- JSEN.goto
- JSEN.break
- JSEN.continue

State Primitives
- JSEN.set
- JSEN.get

Synchronization Primitives
- JSEN.on
- JSEN.getOnStatus
- JSEN.forceCheckOn
- JSEN.sleep

Logging Function
- JSEN.print

Listing 7: JSEN Virtual Language Statements

The main motivation for defining such control statements and synchronization primitives comes from the needs of having a tight control on how programs (implemented as JSEN data structures) are executed. For instance, we want to have the possibility to create a JavaScript program which runs with a specific rate of execution. The statements in Listing 7 allowed us to write programs in a natural way, similar to writing plain JavaScript programs. Additionally, by using statements in Listing 7 we can insert synchronization statements with some events (triggered by other JSEN programs or pure JavaScript one). This gives us the possibility to design and implement a JSEN execution engine that handles JSEN programs like concurrent threads, giving a form of multitasking not available in JavaScript. This choice leads to the implementation of some synchronization primitives as shown in Listing 7. For instance, *JSEN.on()* is a pure JSEN statement that suspends the execution of a JSEN program until a condition is met.

```
1  let var1 = …
2  let var2 = …
3  let jsenTest = [
4    …
5    ()=> console.log( 1 ),
6    JSEN.on( ()=> var1 > var2 ),
7    ()=> console.log( 2 ),
8    …
9  ]
```

Listing 8: Example of usage of jsen_on()

In the example in Listing 8, once the execution of *jsenTest* reaches line 5, it will log 1 to the console, then execute the *JSEN.on()* at line 6. This statement suspends the execution of the *jsenTest* until the value of var1 becomes greater than *var2*. Once the condition is met, the execution will continue by logging 2 as in line 7. This is an example of how the JSEN virtual language can provide new language statements for the implementation of synchronized algorithms.
We now can investigate how JSEN data structure are represented in memory to better understand how they are executed.

3.2 Memory Representation of JSEN data

Since JSEN is defined through a data structure, JSEN programs follows a slightly different process than JavaScript programs (or, in general, JSEN hosting languages). Let us use the example in Listing 9 to see this process.

```
1  function strConcat( str1, str2 ) {
2    return str1+str2;
3  }
4  let i = 10;
5  let jsenTest = [
6    ()=> console.log( 'Start' ),
7    JSEN.if( ()=> i > 1 ),
8      ()=> console.log( strConcat( 'Condition', ' is true' ) ),
9    strConcat( 'This is a string', ' in two parts' ),
10   JSEN.sleep( 1 ),
11   ()=> console.log( 'End' ),
12 ]
```

Listing 9: Example of JSEN with usage of inner and outer functions

In the example of Listing 9 we have the following elements:

- In line 1 the function *strConcat* is defined as static JavaScript function
- In line 4 a variable *i* is defined
- In line 5 a JSEN data structure *jsenTest* is defined

The *jsenTest* data structure is defined within lines 5 and 12. Once we load in JavaScript the source code in Listing 9, the interpreter loads the full source code, and, at the same time performs a sort of compilation step for the JSEN data structure. Once the source is loaded, if we inspect the *jsenTest* variable, it will look like in Listing 10.

```
1  jsenTest[0]: ()=> console.log( 'Start' )
2  jsenTest[1]: { 'name': 'if', 'params': ()=> i > 1 }
3  jsenTest[2]: ()=> console.log( strConcat( 'Condition', ' is true' ) ),
4  jsenTest[3]: 'This is a string in two parts'
5  jsenTest[4]: { 'name': 'sleep', 'params': 1 }
6  jsenTest[5]: ()=> console.log( 'End' )
```

Listing 10: Example of 'compiled' JSEN code

As we can see from Listing 10, the 'compiled' JSEN data structure contains the following types of elements:

- Lines 1, 3 and 6 contains JavaScript statements in form of anonymous functions
- Lines 2 and 5 contains pure JSEN statements. These statements are in the form of JavaScript objects with properties that refer to the name of the JSEN statement ('name') and its parameters ('params')

- Line 4 contains a JSEN comment

Note that the *jsenTest* data structure in Listing 9 makes use of the function *strConcat* in two lines, line 8 and line 9. Those lines are respectively stored in the elements of the array at position 3 and 4 (see Listing 10). It is important to note that at the stage of loading the source in Listing 9, the array *jsenTest* still contains a reference to the function *strConcat* at line 3, since this call is part of the body of an anonymous function. While at line 4 in Listing 10, the function *strConcat* (in line 8 of Listing 9) has been executed by the JavaScript interpreter and therefore the array contains its execution result (a string, as a JSEN comment).

This is something to be considered when coding JSEN structures. Functions directly called in a JSEN data structure like at line 9 of Listing 9 are executed once the JavaScript source code is loaded. This second way of using functions in JSEN data structure can be used for having a sort of macro language, where manipulation of the content of a JSEN data structure can be done at loading time before a JSEN structure is executed. This is one of the different ways in which JSEN could be used for metaprogramming, where the actual content of the JSEN data structure depends on the execution of some functions, executed at loading of the code, or injected at a later time.

Now that the JSEN data structure *jsenTest* is loaded, it can be executed as in the Listing 11.

```
1  JSENVM.run( jsenTest );
```

Listing 11: Execution of a JSEN data structure

This is one possible way to execute JSEN data structures. JSEN is very suited for handling sequential as well as asynchronous code. In the next section we compare different asynchronous methods available in JavaScript with alternatives in JSEN.

3.3 Comparison of Asynchronous Methods

As already mentioned, JavaScript is a single threaded language, a limitation that caused the proposal of different methods for managing asynchronous/ concurrent and parallel computation. The most used choices are callback, Promise, Async/Await statements or workers. The usage of callback [10] is quite straightforward. A function providing an asynchronous execution just needs a parameter which is the callback function to be invoked once the asynchronous execution is completed. Even if this method is particularly simple and easy to implement, code written in this form loses readability and structure. Usage of callback forces developers to split sequential execution into several different functions, making the computational flow difficult to follow. In Listing 15 case 1) we show an example where we want to execute sequentially the following functions: *moveObject*, *rotateObject* and then *displayMsg*. These functions are asynchronous, therefore provide a parameter for a callback (last parameter). Execution is started by *action1*. In Listing 12 case 2, we show an example code that uses anonymous functions. Here the flow looks closer to a sequential flow, however, the code looks more complex to read and maintain due to the different function nesting and brackets.

1) Callback with functions

```
1  function action1() {
2    moveObject( x, y, onMoveDone );
3  }
4  function onMoveDone() {
5    rotateObject( angle, onRotationDone );
6  }
7  function onRotationDone() {
8    displayMsg( 'Action1 done' );
9  }
```

2) Callback with anonymous functions

```
1  function action1() {
2    moveObject( x, y, ()=> {
3      rotateObject( angle, ()=> {
4        displayMsg( 'Action1 done' );
5      });
6    });
7  }
```

3) With JSEN

```
1  let done = false;
2  const action1 = [
3    ()=> moveObject( x, y, ()=> done = true ),
4    JSEN.on( ()=> done == true ), // Suspend until condition true
5    ()=> done = false,
6    ()=> rotateObject( angle, ()=> done = true ),
7    JSEN.on( ()=> done == true ), // Suspend until condition true
8    displayMsg( 'Action1 done' ),
9  ];
```

Listing 12: Example usage of callback compared to JSEN

Through the usage of JSEN it is possible to avoid such problems, keeping the execution flow sequential and minimizing the usage of callbacks (see Listing 12 case 3).

The difference from the case 1 and 2 in Listing 12 compared to case 3 is that with JSEN it is possible to write sequential code that executes asynchronous calls, each one after the other. In JSEN we can use JSEN.on() to suspend execution until the condition specified as parameter becomes true. We then use the callbacks of asynchronous functions (moveObject, rotateObject) to change the value of the condition used in JSEN.on() to continue computation. Here, the contribution of JSEN is the possibility to write sequential code that controls asynchronous calls at the same time.

In JavaScript, usage of Promises [29] is an alternative way for dealing with asynchronous calls

while keeping a more readable flow. Promises act as proxy for callbacks, which create the possibility of expressing asynchronous calls into a more readable and sequential way. Promises tries to solve the problems we found with callback (readability, nesting of functions, …). The example of case 1 in Listing 12, with the usage of Promises, can be written as in Listing 13.

```
1  function action1() {
2    moveObject( x, y )
3    .then( ()=> rotateObject( angle ) )
4    .then( ()=> displayMsg( 'Action1 done' ) );
5  }
```

Listing 13: Example usage of Promises

However, note that now, both asynchronous functions *moveObject* and *rotateObject* must be rewritten, to make them return a Promise as result. This allows to call the "*.then()*" method for continuing computation. This is possible when we are the owners of the functions, otherwise (in case of 3rd party libraries) a wrapper function using Promises must be written for each function we want to use. Moreover, in this new scenario, we should always check if a function we want to use implements Promises or not. Here the contribution of JSEN is in the possibility to use asynchronous functions as they are, just calling them inside a JSEN program in a sequential way as shown in Listing 12.

```
1  async function action1() {
2    await moveObject( x, y );
3    await rotateObject( angle );
4    displayMsg( 'Action1 done' );
5  }
```

Listing 14: Example usage of Async/Await

A recently added set of statements to JavaScript are the Async/Await [9]. These two statements are based on Promises. This means they should be applied to functions that returns Promises, if not, JavaScript will generate a default Promise to be returned by the function. This makes execution of asynchronous code much cleaner from a syntactical point of view. Here in Listing 14 we show a code based on the example of Listing 13.

Usage of Async/Await improves substantially readability and maintainability. However, here also, it is necessary to modify every function that needs to use such statements, particularly because the usage of 'await' can only be done in a function declared as 'Async'. Moreover, in both usage of Promises and Async/Await, it is easily possible to lose control over the functions that need synchronization and the ones that do not, introducing bugs not easy to locate [1]. One of the main difficulties here is to decide when to use 'async', a choice that may have to be taken line per line in some cases. Another consequence of the usage of Async/Await is that the main JavaScript thread is suspended for the execution of each Async function. In this case the contribution of JSEN is that it does not require to modify functions. More importantly, JSEN does not suspend the execution of the JavaScript main thread in handling asynchronous code. This allows JavaScript to still use the main thread for execution of other JavaScript code.

A true parallel execution can be reached by the usage of workers [12]. In JavaScript, a worker is a program executed in a separate thread, running in parallel to the caller context. In terms of coding, maintainability, and parallelization, this is the best solution among the one we reviewed. Programs executed in workers are just normal JavaScript programs, and do not interfere with computation done in caller contexts. However, usage of workers introduces some limitations in terms of communication, data and library sharing. Workers are executed in a separated context than the calling program, therefore they cannot access data instances available in the caller's context and cannot use libraries that depend on such data. Communication between a worker and its calling context is done via messages.

On the one side this can be considered a good practice for protecting mutual access on data, however, on the other side it introduces a strong limitation on how a worker and its calling context can interact. In this case, JSEN does not provide true parallelism (JSEN execution is done in time sharing with calling context). However, JSEN provides the possibility to access any data structure or library of the calling context. Moreover, JSEN runs in time sharing with JavaScript and the execution of each single JSEN statements is atomic. Atomicity comes from JavaScript being single threaded. Therefore, handling of mutual access of data becomes easy compared to the usage of mutex/semaphores/locks.

In the next paragraph we see other ways which also includes the possibility for concurrent execution of multiple JSEN programs.

3.4 Concurrency with JSEN

The granularity at which JSEN data structures stores statements gives the possibility of implementing a form of concurrent multitasking among JSEN programs. This concurrency is particularly useful in JavaScript, since JavaScript is a single threaded language and therefore, beside the usage of "workers" [12], parallel execution of functions (within the same process) is not possible. In the community, this limitation has been circumvented by the usage of several methods for asynchronous programming [22], as we have just seen

with the use of callback, implementation of Promise, or the Async/Await statements.

With JSEN we provide an alternative way to handle asynchronous and concurrent tasks. In our view, the usage of JSEN data structures makes creation and maintenance of asynchronous and concurrent tasks easy to write and to maintain. Concurrency in JSEN is achieved through a virtual machine. Such a machine can take several JSEN data structures and executes them according to a concurrent policy. To illustrate this concept, we have implemented JZENVM, a small virtual machine that implements the core concepts, supporting a basic set of pure JSEN statements (a sub-set of the ones showed in Listing 7).

In Listing 15 we give the pseudocode of the JZENVM virtual machine. The *JZENVM_run* function in Listing 15 takes a variable number of parameters. Each parameter should be a JSEN data structure. The function starts by creating an execution context for each parameter where the attribute 'code' points to the n-th parameter (JSEN data structure). The *JZENVM_runContext* function takes the created contexts and executes each of them until they all terminate their execution. Execution of contexts uses a round-robin scheduler [21] and rely on a timer for waking-up execution in case contexts are suspended (e.g. through *JSEN.sleep*, *JSEN.on*, …).

```
function JZENVM_run( all parameter ) {
  for each parameter
    create new context
      set properties: executionStatus, code, pc, caller
  JZENVM_runContext( all context );
}
function JZENVM_runContext( all context ) {
  while not all context are terminated
    for each context
      if context is not terminated or suspended
        get next context's statement and increment pc
        switch( type of statement )
          case anonymous function → call it
          case array → switch context code to array (sub-block)
                       and store current code in caller
          case object → this is a JSEN.* statement
            switch( object.name )
              case 'if' → check condition and update pc
              case 'sleep' → set context to suspended
                             and start timer for wakeup
              case 'label' → assign pc to label value
              case 'goto' → set pc to label value
              case 'print' → print parameter to console
            otherwise
              skip statement
      if context is terminated and caller context is not empty
        restore caller context code, pc, …
}
```

Listing 15: Pseudo code for the JZENVM virtual machine

The usage of a scheduler in the *runContext* function, allows concurrent execution of different JSEN data structures in time-sharing. If no statement of JSEN data structures implements an infinite loop, the runContext function can execute one statement from each context at a time, giving the possibility to all JSEN data structures, step by step, to progress their computation.

This simple and compact version of a JSEN virtual machine is a good example for understanding how JSEN data structures can be executed concurrently. We also implemented JSENVM, a more complete JSEN virtual machine supporting all statements shown in Listing 7.

Here in Listing 16 an example of three JSEN threads: *printNumbers*, *printUpLetters*, *printLowLetters*. Each of them computes different values, printing them to the console. This example shows how the *JSENVM_run()* method executes them concurrently.

```
const JSEN = require( 'JSEN' );
const JSENVM = require( 'JSENVM' );
let number;
const printNumbers = [
  JSEN.for( 'i', 0, 3 ),
  [
    JSEN.get( 'i', (value)=> number = value ),
    ()=> console.log( number ),
  ],
];
let upLetter = 'A'.charCodeAt( 0 );
const printUpLetters = [
  JSEN.for( 'i', 0, 3 ),
  [
    ()=> console.log( String.fromCharCode( upLetter ) ),
    ()=> ++upLetter,
  ],
];
let lowLetter = 'a'.charCodeAt( 0 );
const printLowLetters = [
  JSEN.for( 'i', 0, 3 ),
  [
    ()=> console.log( String.fromCharCode( lowLetter ) ),
    ()=> ++lowLetter,
  ],
];
JSENVM.run( printNumbers, printUpLetters,
printLowLetters );
```

```
> node concurrentExample1.js
A
a
0
B
b
1
C
c
2
```

Listing 16: Example of different JSEN threads executed by JSENVM

JSENVM beside the *run()* method (used for an handy execution of threads), provides a more complete API for handling JSEN threads. Listing 17 shows an excerpt of the API exposed by JSENVM.

```
1  newThread( name, code, … )
2  startThread( nameOrList )
3  stopThread( nameOrList )
4  suspendThread( nameOrList )
5  wakeupThread( nameOrList )
6  renewThread( nameOrList )
7  removeThread( nameOrList )
8  isThreadReady( name )
9  isThreadRunning( name )
10 isTreadSuspended( name )
11 isThread…
12 setBreakPoint( name, condition, action )
13 addThreadJoin( nameOrList, joinFunction )
14 removeThreadJoin( joinFunction )
15 …
```

Listing 17: Extract of part of JSENVM API

Table 1: Performance of JSEN compared to pure JavaScript algorithms

Task	Javascript execution time (1000 iter.)	JSEN execution time (1000 iter.)	Factor slower
20x20 Matrix multiplication	42,0ms	3.915,5ms	93x
Bubble sort of array of 60 elements	6,7ms	1.103,5ms	164x
Prime factor of 100x 2-digit numbers	13,2ms	1.065,8ms	79x
Simple search of a 20-string in a 100-string	6,3ms	137,4ms	22x
Multiplication of 2 numbers	2,8ms	38,0ms	14x

Table 2: How JSEN improve JavaScript

Desirable properties	Pure JavaScript	JSEN
Homoiconicity	Very limited	Extended
Serialization of code	Limited	Extended
Virtual Language	No	Yes
Concurrency	Webworker (data transfer)	JSENVM (full data access)
Asynchronicity	Nested code	Linear code
Metaprogramming	Limited	Extended
Performance	High	Low

The JSENVM virtual machine implements the following functionalities for managing JSEN threads:

- Set and handle JSEN threads life cycle (creation, execution, …)
- Check JSEN thread status (ready, running, …)
- Handle thread-join functions
- Debugging functions (step-by-step, breakpoints, …)
- Supports of several pure JSEN statements (see Listing 7)

The next section describes now the most important properties exposed by JSEN.

4 Properties of JSEN

Given the close similarities between JSEN and JSON, JSEN carries several properties and capabilities of JSON. JSEN, as JSON, is a text-based data format, it is compact and lightweight (in relation to the data content) and JSEN has a relatively strong connection with the hosting language, enabling a full reuse of the hosting language. Because of that, unlike JSON, JSEN is language specific. Looking at JSEN from a JavaScript perspective, it exposes the following properties:

Extended Homoiconicity: JSEN extends the concept of "code as data"; algorithms/functions in a program implemented in JSEN provides itself as a data structure suitable for storing code, execute code and/or manipulate it programmatically by the program itself. This is an extension of the homoiconic [6] and metaprogramming [23][5] capabilities of JavaScript, bringing it closer to what languages like Lisp [26], SmallTalk [7] or Tcl [28] can do. It introduces a finer grain control of JavaScript language statements as well as it enables the creation of virtual languages. This opens the possibility of symbolic programming [30], self-modifying code, learning and other metaprogramming paradigms.

Serialization: similarly to *JSON.stringify()/parse()*, JSEN structure can be serialized into a single string with *JSEN.stringify()/parse()*. However, at the moment, we did not implemented a safe *parse()* function. We are still parsing JSEN strings through the JavaScript *eval()* system function. Nevertheless, like in JSON, it is possible to serialize a JSEN data structure, send it to another program, which can deserialize and continue the computation in another context.

Performance: we have been benchmarking the executions of algorithms written in JSEN and their equivalent in pure JavaScript. The results are summarized in Table 1. We used JavaScript on a Windows 10 machine with Node.js Version 12.16.1 on an IntelCore i7 2.5GHz. As expected, JSEN implementations are much slower than pure JavaScript ones. This is a direct consequence of JSEN being built on top of JavaScript and making extensive use of functions and anonymous functions. herefore, JSEN is best used for what it was designed for, namely readability of code in asynchronous tasks, easy parallelization using time sharing, virtualization or metaprogramming. However, it is always possible to convert back and forth between JSEN and pure JavaScript. Moreover, the computationally expensive part of an algorithm should be written in pure JavaScript and called from JSEN if needed. The slowdown factor is, as can be observed in the 3 first algorithms of Table 1, mostly impacted by the number of nested loop and the total number of iterations.

In the cases where performance of a JSEN program is an issue, it is possible to either transpile JSEN code into native JavaScript code or to compile it to WASM. As we mention in section 5 this topic is not covered here in this introductory paper but is possible.

Virtual Languages Support: JSEN introduces a data structure that makes the creation of new JSEN statements very easy. In this way, new language-like features (virtual language [3]), control execution flow or macro languages can be created using the same hosting language. Even full embedded domain specific languages (DSL [13]) could be implemented with that. Advantages in terms of domain analysis, rapid prototyping, portability, and maintenance can be easily provided. For instance, the creation of a virtual language for a specific project can offer a more expressive way for encoding it. Moreover, when necessary, porting the resulting code to different languages may become a relatively easy code generation task (thanks to metaprogramming).

Concurrency: JSEN introduces a new way to execute concurrent functions (time sharing) together with the main JavaScript thread (hosting language). This allows the execution of several asynchronous functions as they would be running in parallel threads. Unlike the case of workers (see the previous paragraph), JSEN functions have access to all data structures and libraries used in the calling context. Furthermore, each single statement of a JSEN data structures is executed atomically (cannot be interrupted). This makes handling of synchronization between different JSEN functions much easier than threading done at a lower level, where, each single statement could be suspended in the middle of its execution, leading to the needs of a more explicit handling of atomicity by developers.

The previous list show some of the properties of JSEN and it not meant to be exhaustive. The next section will briefly summarize other important aspects of JSEN not covered in this paper.

5 Further Concepts in JSEN

This paper is meant as a JSEN introduction by covering some of its basic characteristics and properties. We showed it as a data structure as well as how it can be used to execute functions, and how it can be used to handle asynchronous computation in the scope of the JavaScript language. JSEN is meant to be used in any scope where these characteristics are relevant. For instance, where a more expressive language than JavaScript is necessary, JSEN can be used to create appropriate language extensions (see *JSEN.on()* as one of those cases). In the scope of concurrent or asynchronous execution of code, JSEN provides a flexible alternative (avoid JavaScript call-back hell). Several additional topics of JSEN on which we have been working have not been covered in this paper:

- The manipulation of JSEN data structures (metaprogramming).
- A description of all pure JSEN statements and how to use them.
- The usage of JSENThreadClass and active-objects.
- A more in-depth review on how standard JavaScript code can be transpiled into JSEN and vice versa.
- The conditions on which JSEN can be transferred between processes.
- The portability of JSEN to other languages (we already tested on Java, Python, Matlab, see examples in the Git repository).
- How to extend/redefine virtual languages.
- A closer analysis on JSENVM API and ways to manage concurrent tasks.
- Debugging with JSENVM and a web-based debugger.

6 SUMMARY AND CONCLUSIONS

In this article, we introduce JSEN as a new data format for representing executable code (as counter part of JSON data format). We have shown how it can be used to store algorithms/functions/programs, which could be exchanged between processes or used to be executed like functions. We summarized the properties of JSEN in Table 2. We have shown how JSEN introduces a higher level of homoiconicity in the hosting language, enhancing the possibility of manipulating code in the context of metaprogramming.

As we have seen JSEN introduces the possibility to create virtual languages. In that scope we have shown how a JSEN data structure gets 'compiled' and how execution can be done via a virtual machine. We reviewed JZENVM, a basic virtual machine that shows the main principle of JSEN concurrent execution and introduced JSENVM, a more complete virtual machine. Given that concurrent and asynchronous computation are important in JavaScript we have compared how asynchronous and concurrent functions can be done in JSEN compared to other available alternative for asynchronous programming like callback, Promises, Async/Await and workers. Clearly there are certain advantages in using JSEN.

In terms of metaprogramming, it gives a finer grain access to statements of a function, which can be more easily introspected and manipulated in a programmatic way. As well, using JSEN as a mean to create concurrent computation in a single threaded language, JSEN has its advantages in approaching several issues that appears in handling asynchronous code. Similarly, as workers, it is possible with JSEN to execute functions concurrently, avoiding blocking the JavaScript main thread.

ACKNOWLEDGEMENT

This work was financed by Honda Research Institute GmbH. The authors want to thank all the reviewers for the very constructive and rich comments as well as Kálmán Graffi, Cristian Sandu, Andrei Huțucă for early discussions on the concept and comments on the paper, Cătălina Ioan, for her review and help on the development of the virtual language and debugging tool, and Siddhata Naik for her support on open-sourcing JSEN.

REFERENCES

[1] A. Agarwal, "How to escape async/await hell", https://www.freecodecamp.org/news/avoiding-the-async-await-hell-c77a0fb71c4c/, Accessed 29 April, 2021

[2] M. Bolin, "Closure: The definitive guide: Google tools to add power to your JavaScript", O'Reilly Media, Inc, 2010

[3] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, K. Olukotun, "Language virtualization for heterogeneous parallel computing", *ACM Sigplan Notices*, pp. 835-847, 45(10), 2010

[4] W. Cunningham, "Homoiconic languages", https://wiki.c2.com/?HomoiconicLanguages, Accessed 29 April, 2021

[5] R. Damaševičius, V. Štuikys, "Taxonomy of the fundamental concepts of metaprogramming", *Information Technology and Control*, 37(2), pp. 124-128, 2008

[6] T. Davies, "Homoiconicity, Lazyness and First-Class Macros", CiteSeerX, pp. 2-6, 2009

[7] W. De Meuter, (Ed.), "Advances in Smalltalk: 14th International", Smalltalk Conference, ISC 2006, Prague, Czech Republic, September 4-8, 2006, Revised Selected Papers, vol. 4406, pp. 14-17, 2006

[8] B. Dias, "Callback hell, promises, and asynch/ await", https://blog.avenuecode.com/callback-hell-promises-and-async/await, Accessed 29 April, 2021

[9] I. Elliot, "JavaScript async: Events, callbacks, promises and async await", I/O Press, 2017

[10] K. Gallaba, A Mesbah, and I. Beschastnikh, "Don't call us, we'll call you: Characterizing callbacks in JavaScript", ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1-10, October 2015

[11] A. Goldberg, R. David, "Smalltalk-80: The language and its implementation", Addison-Wesley Longman Publishing Co., Inc., 1983

[12] I. Green, "Web workers: Multithreaded programs in javascript", O'Reilly Media, Inc., 2012.

[13] F. Hermans, M. Pinzger, a. van Deursen, "Domain-specific languages in practice: A user study on the success factors". In: Schürr A., Selic B. (eds) Model Driven Engineering Languages and Systems. MODELS 2009. Lecture Notes in Computer Science, vol 5795, pp. 1-13, Springer, Berlin, Heidelberg, 2009

[14] E. M. Hvidevold, "Majaho", https://github.com/ emnh/majaho, Accessed 29 April, 2021

[15] JSONIntro, "Introducing JSON", http://www.json.org, Accessed 29 April, 2021

[16] JSONRef, "The JSON data interchange format", ECMA International, October 2013

[17] JSONSyntax, "Standard ECMA-404 -The JSON data interchange syntax", 2nd edition, December 2017, https://www.ecma-international.org/publications/standards/Ecma-404.htm, Accessed 29 April, 2021

[18] JSSyntax, "JavaScript Object Literal Syntax", https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Object_literals, Accessed 29 April, 2021

[19] A. Korpi, C. Meadows, W. Pimenta, "EsLisp", https://github.com/anko/eslisp, Accessed 29 April 2021

[20] R. G. Lavender, D. C. Schmidt, "Active object - An object behavioral pattern for concurrent programming", CiteSeerX, pp. 2-8, 1995

[21] P. G. López, J. S. V. Martínez, G. D. and Reyes, "Concurrent real-time task schedulers: A classification based on functions and set theory", *Computacion y Sistemas*, pp.809-820, 2015

[22] M. C. Loring, M. Mark, L. Daan, "Semantics of asynchronous JavaScript.", In Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages, pp. 51-62., 2017

[23] A. Ludwig, D. Heuzeroth,, "Metaprogramming in the Large", In International Symposium on Generative and Component-Based Software Engineering, pp. 179-188, October 2000,

[24] S. R. Maxwell, "Experiments with a coroutine execution model for genetic programming", In Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, pp. 413-417, 1994

[25] C. N. Mooers, L. P. Deutsch, R. W. Floyd, R. "Programming languages for non-numeric processing-1: TRAC, a Text Handling Language", in Association for Computing Machinery, NY, USA, pp. 229-246, August 1965.

[26] J. Newton, D. Verna, M. Colange, "Programmatic manipulation of common Lisp type specifiers", In European Lisp Symposium, pp. 1-3, 2017.

[27] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, "Comparison of JSON and XML data interchange formats: A case study", Caine, pp. 157-162, Nov. 2009

[28] J. K. Ousterhout, "Tcl: An embeddable command language", University of California, Berkeley, Computer Science Division, 1989.

[29] D. Parker, "JavaScript with Promises: Managing asynchronous Code", O'Reilly Media, Inc., 2015

[30] N. Rochester, "Symbolic programming", Transactions of the IRE Professional Group on Electronic Computers, EC-2(1), pp. 10-15, 1953

[31] G. Steele, G., "Common LISP: the language", Elsevier, 1990

[32] C. Wang, S. Hasler, M. Mühlig, F. Joublin, C. Ceravola, J. Deigmöller, F. Fischer, "Designing interaction for multi-agent cooperative system in an office environment", ACM/IEEE International Conference on Human-Robot Interaction, pp. 668-669, March 8, 2021.

[33] XMLApp, "XML applications and initiatives", http://xml.coverpages.org/xmlApplications.html, Accessed 29 April, 2021

[34] XMLAlternative, "XML Alternatives", http://web.archive.org/web/20060325012720/www.paul t.com/xmlalternatives.html, Accessed 29 April, 2021

[35] XMLCritics, "XML: The angle bracket tax", https://blog.codinghorror.com/xml-the-angle-bracket-tax/, Accessed 29 April, 2021

[36] XMLRef, "Extensible Markup Language (XML) 1.0", https://www.w3.org/TR/1998/REC-xml-19980210, Acessed 29 April, 2021

[37] S. Zunke, V. D'Souza, "JSON vs XML: A comparative performance analysis of data exchange formats", IJCSN International Journal of Computer Science and Network, 3(4), pp. 257-261, 2014

AUTHOR BIOGRAPHIES

Antonello Ceravola: He studied Computer Science at the University of Pisa, Italy. He worked in the field of IT software for five years dealing with multimedia systems, large scale software infrastructure for telecommunication systems, multi-tier applications and workflow engine for process management systems. From 2001 he joined Honda Research Institute Europe, Germany, currently Principal Scientist. His research interest includes software components, middleware, languages, large-scale systems, integration environments, autonomous driving system and artificial intelligence.

Frank Joublin: He received a European Ph.D. degree in neurosciences from the University of Rouen (France), in 1993. From 1994 to 1998 he was postdoctoral research fellow at the Institute für Neuroinformatik, university of Bochum, Germany. From 1998 to 2001 he was customer project manager at Philips Speech Processing Aachen. Since 2001, he is principal scientist at the Honda Research Institute Europe, Germany. His research interests include developmental robotics, semantic acquisition, data mining and artificial intelligence.