Competitors or Cousins? Studying the Parallels between Distributed Programming Languages SystemJ and IEC61499

Roopak Sinha
Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
rsinha@aut.ac.nz

Valeriy Vyatkin
Luleå University of Technology,
Luleå, Sweden and
Aalto University, Helsinki, Finland
vyatkin@ieee.org

Zoran Salcic, Hee Jong Park
Electrical & Computer Engineering
University of Auckland
Auckland, New Zealand
z.salcic@auckland.ac.nz;
hpar081@aucklanduni.ac.nz

***Abstract*—We face a glut of languages for programming distributed software today. However, only a few languages have proven their potential with wider practical use in different domains of computing. We picked two such languages, meant for different domains, to see if they could cross-pollinate and enrich one another. Specifically, we chose SystemJ, a language to program distributed embedded systems, and IEC61499, the next generation standard for distributed industrial automation control software. Unsurprisingly, we found similar structures and artifacts between the two. We also found significant differences mainly due to differing domain-specific requirements. This comparison leads to observations and guidelines for improving both languages, and we discuss directions towards an "ideal" distributed software programming language.**

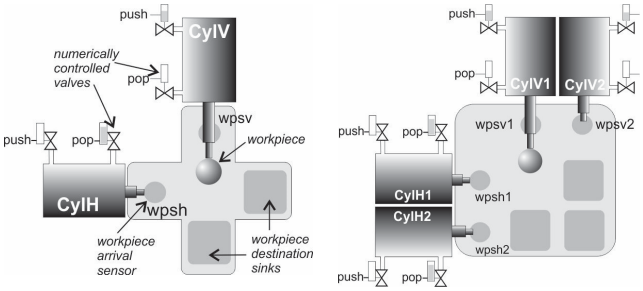
Keywords—IEC 61499, SystemJ, distributed programming, concurrent programming,

I. INTRODUCTION

The age of stand-alone, single process-single machine computing gave way to distributed computing decades ago. Languages to program distributed software systems exist since the late 1970s [1], with new ones appearing at an alarmingly high rate. This glut highlights how different domains of computing have embraced distributed design, while also suggesting that the current offerings are not ideal.

Our goal is to contribute towards the direction of unifying distributed languages by studying existing languages that have proven their potential in the real world. This article is a first step in the process where we study two significant and critically important domains of computing: embedded systems and industrial automation. While a number of languages for distributed design in these domains exist [2], [3], [4], we chose *system-level* [8] design languages SystemJ [5] and IEC61499 [6, 7] due to their popular use in their respective domains. SystemJ, based on Java, uses a Globally-Synchronous, Locally Synchronous (GALS) [9] execution semantics for distributed embedded programs. IEC 61499 is the successor of IEC 61131-3 [10], and uses reusable visual constructs called function blocks to program complex industrial automation controllers.

We study both languages by implementing a small distributed system, based on a family of reference examples with increasing complexity. Fig. 1a shows a system with two cylinders: a horizontal cylinder (H) and a vertical cylinder (V). Each cylinder set up consists of a linear motion pusher.



a) 2-cylinders system. b) 4 cylinders.

Fig. 1: Reference example

Once a workpiece is placed in front of the pusher (that is detected by the WPS sensor), the pusher must then move the workpiece to the destination sink and retract the pusher to its initial state. The dual cylinder system in Fig. 1a requires additional coordinating logic to avoid potential collisions between the pushers. We choose a ring-token approach [11], where a cylinder fires only when a work-piece has arrived *and* it holds the token. A cylinder passes the token to the other cylinder after moving a workpiece and retracting the pusher or when no workpiece is present. This simple assembly can be extended further to include 4 (Fig. 1b), 6, 8 or more cylinders to model more complex scenarios. However, we persist with the 2 and 4-cylinder examples in this paper because it is sufficiently rich for studying the two languages and also lends itself to easy illustration.

The illustrative example represents the requirements of flexible manufacturing systems, in which distribution of control across mechatronic actors is seen as a means to improve their re-configurability and flexibility. In this context, a single pusher can be seen as a simplified model of a machine, pre-programmed to perform one operation, while a composition of (2, 4 or more) cylinders, is a model of a manufacturing system, composed of autonomous machines, requiring some coordination. Therefore, the features of programming languages that support this (mechatronic plug and play) composition efficiently are of highest interest.

We program this system in both SystemJ and IEC 61499 function blocks, and then compare the two implementations to highlight the similarities and differences between the two languages. However, comparing these two languages is a secondary contribution. We primarily look at the causes of the similarities and differences (both domain-independent as well as domain-dependent) between the two languages to bring out features that are best suited for the designing complex distributed systems.

Having set the scene, we now present the details. Sec. II and Sec. III describe the two languages and their execution. Sec. IV compares the two languages, and makes recommendations on how we can enrich the two. Concluding remarks and future directions appear in Sec. V.

II. SYSTEMJ

SystemJ [12] is a system-level programming language based on the *Globally Asynchronous Locally Synchronous* (GALS) Model of Computation (MoC). The language provides features suitable for designing complex concurrent and distributed reactive systems and naturally supports the development of many new classes of embedded applications such as smart spaces, home and building automation robotics, distributed control and collaborative systems. By being based on the GALS MoC, the language supports programming entities and abstractions that allow easy and safe handling of synchronous and asynchronous concurrency, both being common and important for structuring most of the targeted applications. In addition, through integration of Java, SystemJ is capable of handling traditional object-oriented data abstractions and performing complex sequential data computations. Besides the high-level abstractions provided for describing different types of concurrency, SystemJ supports abstractions to engage physical events (sensing and actuation) and communications over different hardware platforms without concerns on the nature of the execution platform and communication mechanisms, thus abstracting one of the major hurdles of heterogeneity of platforms. Having a formal operational semantics, SystemJ ensures the correctness of the implemented software behaviors during the design process.

A. Syntax and Structure

A single SystemJ program, called a *system*, contains one or more asynchronous concurrent behaviors encapsulated in the language entity called a *clock domain*. Fig. 2 visually shows the structure of a SystemJ program for the two-cylinder example in Fig. 1a. The system contains two clock domains - CD1 and CD2.

Each clock domain can be further refined into one or more synchronous concurrent sub-behaviors, which are called *reactions*. Reactions can be seen as concurrent programs scheduled on their parent clock domains. As shown in Fig. 2, clock domains CD1 and CD2 contain reactions H and V respectively, which control the horizontal and vertical cylinders respectively. Since reactions are coupled strongly with the SystemJ execution semantics, we provide the details of H and V after presenting the semantics in Sec. II-B.

SystemJ provides two abstract objects for communications between reactions: signals and channels. These abstractions hide the underlying physical devices and communication protocols from the system designer. *Channels* allow uni-directional point-to-point delivery of Java primitive or object variables between reactions in *different* clock domains.

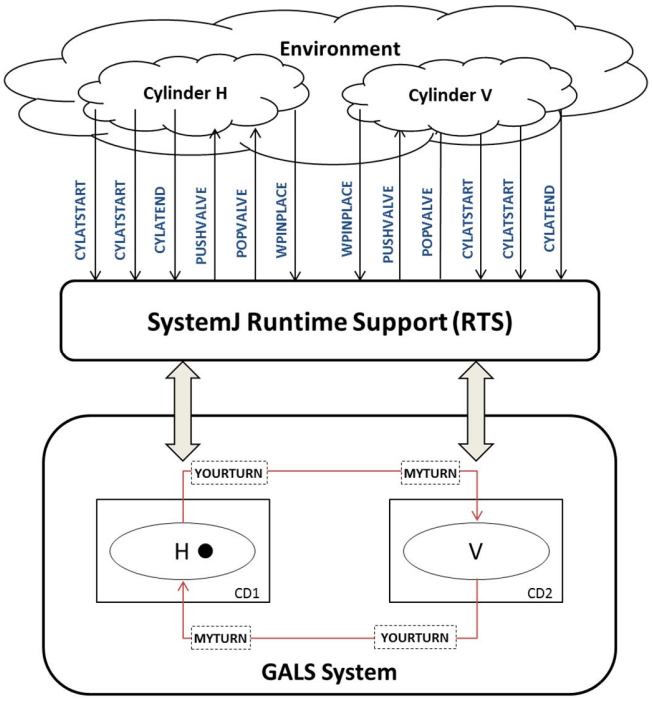


Fig. 2: Graphical presentation of a SystemJ program.

E.g., Fig. 2 shows an output channel `YOURTURN` in reaction H connected to an input channel `MYTURN` of reaction V. On the other hand, *signals* allow communication between reactions within a clock domain. Each signal has a *status*, which is either *present* or *absent*. In addition, *valued* signals carry values (a Java object or primitive type) along with their status. Signals also serve as the primary communication mechanism between reactions and the external environment (e.g. signals WPINPLACE, POPVALVE etc. in Fig. 2). Such uni-directional *interface* signals uniformly represent physical device events such as timer timeouts, sensor and actuator IOs, and communication events. The semantics of channel and signal based communications is explained in Sec. II-B.

B. SystemJ Execution Semantics

Reactions are the elementary execution blocks in SystemJ. A reaction executes *synchronously* [13], in logical time instants called *ticks*. Ticks originate from a logical clock. During each tick, a reaction samples the environment, executes until the next *tick boundary*, and emits outputs. During different ticks a reaction may execute different instructions and hence

ticks may take varying times to complete. The sequence of ticks therefore represents a *logical clock*.

Fig. 3 shows the SystemJ program for the two-cylinder example in Fig. 1a. The inputs and outputs of the reactions `ControllerWithToken` and `Controller` are shown in lines 2–8 and 24-30 respectively. Each reaction implements the logic of a single cylinder controller, and is instantiated to create the two controllers for horizontal and vertical cylinder as clock domains (lines 45–46). The `ControllerWithToken` reaction first checks for presence of the interface signal WPINPLACE (line 10), which indicates the appearance of a work piece. When the signal is present, the reaction turns on the push-valve of its corresponding cylinder to push the work piece until the signal CYLATEND (indicating the cylinder is fully extended) is read (lines 11–13). Next, the controller retracts the pusher by turning on the POPVALVE and waits until it is fully retracted by awaiting the CYLATSTART signal (lines 14–16). The controller then passes the token using the output channel `YOURTURN` (line 18). It then awaits a token on channel `MYTURN` ensuring that cylinders do not move simultaneously. `MYTURN` is a channel and reaction blocks (line 19) the reaction blocks until the token is received. This behavior repeats forever. The logic of `Controller` reaction is identical to `ControllerWithToken`; except it first awaits a token on the channel `MYTURN` (line 32) before the cylinder can be fired.

```
1  reaction ControllerWithToken(:
2          input signal WPINPLACE,
3          input signal CYLATSTART,
4          input signal CYLATEND,
5          output signal PUSHVALVE,
6          output signal POPVALVE,
7          output boolean channel YOURTURN,
8          input boolean channel MYTURN){
9      while(true){
10         present(WPINPLACE){
11             abort(CYLATEND){
12                 sustain PUSHVALVE;
13             }
14             abort(CYLATSTART){
15                 sustain POPVALVE;
16             }
17         }
18         send YOURTURN(true);
19         receive MYTURN;
20         pause;
21     }
22 }
23 reaction Controller(:
24         input signal WPINPLACE,
25         input signal CYLATSTART,
26         input signal CYLATEND,
27         output signal PUSHVALVE,
28         output signal POPVALVE,
29         output boolean channel YOURTURN,
30         input boolean channel MYTURN){
31     while(true){
32         receive MYTURN;
33         present(WPINPLACE){
34             abort(CYLATEND){
35                 sustain PUSHVALVE;
36             }
37             abort(CYLATSTART){
38                 sustain POPVALVE;
39             }
40         }
41         send YOURTURN(true);
42         pause;
43     }
44 }
45 H(..)->ControllerWithToken
46 V(..)->Controller
```

Fig. 3: SystemJ program for the 2-cylinders system.

All reactions within the same clock domain are driven by the same tick, and hence execute in a *lock-step*. In fact, any clock domain can be considered a single synchronous reaction comprising multiple (sub) reactions that execute synchronously. Since intra-clock domain synchronous communication may have causal loops [13], SystemJ uses *delayed semantics*, similar to [14], where any exchange of information between reactions happens only at tick boundaries. In other words, signals emitted by one reaction in tick *i* are *broadcast* to and seen by other reactions in tick *i*+1. Delayed semantics provide several benefits: the sequence of execution of reactions in a clock-domain does not affect overall program behavior, clock domains execute deterministically, and programs are non-causal by construction.

Different clock domains are driven by their own logical clocks and hence execute asynchronously. Here, we cannot rely on signal broadcasting with its limited (one clock tick) lifetime that is neither reliable not sufficient for inter-clock domain communication. Therefore, SystemJ provides persistent communication using *channels* to exchange data (messages) between reactions of different clock domains. Channels implement *rendezvous*-style message-passing mechanism adopted from Communicating Sequential Process (CSP) [15], which ensures the delivery of the message. The sender and receiver reactions employ the SystemJ statements *send* and *receive,* respectively to perform channel communication. Channel communications are blocking. The sender reaction blocks after a *send* until the message is received by the receiver. Similarly, a receiver reaction blocks after a *receive* if there is no message available. The rendezvous-based synchronization happens when channel *receive* is successful, and then both the sender and receiver reactions can execute further. It is possible to re-write the program with dedicated reactions for channel communications in both clock domains to eliminate potential infinite blocking of the entire clock domains if rendezvous is not successful.

C. Case study implementation

The user maps interface signals and channels of a SystemJ program onto physical signals and communication mechanisms using an additional *configuration* file. All signals in the SystemJ program for the cylinder example (illustrated in Fig. 2) are interface signals exchanged between the two reactions and the environment. For example, these signals may bind to individual IP ports on the local machine, and this information is contained in the mapping file. This allows the SystemJ program to seamlessly exchange information with suitable plant-modeling software, or even a driver interface to an actual plant (of the two cylinders). The configuration file also contains the mapping of channels MYTURN and YOURTURN of the horizontal cylinder controller to channels YOURTURN and MYTURN of the vertical cylinder controller respectively, thus allowing implementation of this communication using shared memory

(if on local machine) or any other communication mechanism, e.g. over a network.

D. Executing Distributed SystemJ programs

The SystemJ compiler [12] translates SystemJ code into Java. The Java code can then execute on various JVM enabled execution platforms or Java processors [16] depending on the available platform.

Platform dependent libraries, such as initialization code and underlying communication method for SystemJ channels and signals handling are provided and maintained separately by SystemJ Run Time Support (RTS). The RTS abstracts away the physical devices, such as sensors and actuators over different networking mechanisms and communications over different protocols such as TCP, UDP, and Bluetooth, and allows the SystemJ program to use signals and channels instead. SystemJ RTS has been ported on various embedded and mobile platforms, including the Java standard and micro editions, Android devices, and on resource constrained platforms such as Squawk VM for SunSPOT sensor nodes [16], [17] and LeJOS for Mindstorms robots [18]. The RTS automatically implements the mappings (contained in the configuration file) during program execution.

III. IEC 61499

A. Syntax and Structure

A program in IEC61499 (called application) contains instances of reusable modules called *function blocks*. Each function block has a well-defined *interface*, which explicitly states the input events, output events, input variables and output variables, of the block. Fig. 4 shows the interface of the function block `Control`. The interface contains the input events INIT, WP, REQ, and the output signals INITO, CNF, COMPLETE and DROP. Similarly, it contains the input variables QI, START and END (Boolean) and the output variable FORCE (of type real). Input (output) events are *associated* to input (output) variables using *associations*. E.g., output event CNF is associated to the output variable FORCE in Fig. 4a. Events can be seen as transitory pulses or messages, while variables are persistent data. Due to associations, a block updates the value of a variable only when an event associated with that variable arrives (or is sent). Other types of interface elements are adapter sockets and plugs. These are used to enable adapter connections between function blocks that are bundles of event and data connections encapsulated into one "thick cable", the use of which will be illustrated in Figs.6 and 7.

A *basic* function block is the elementary component of IEC 61499, and consists of an interface, *local declarations*, and an *execution control chart* (ECC) (Fig.4, right side). Local declarations consist of *internal* variables of the block and *algorithms* (written in standard languages such as C or Java). Algorithms may operate over internal, input and output variables of the block. The basic block Control, whose interface is shown in Fig. 4, also contains a single internal variable `oktogo` and algorithms *FWD*, *STOP*, and *BACK*.

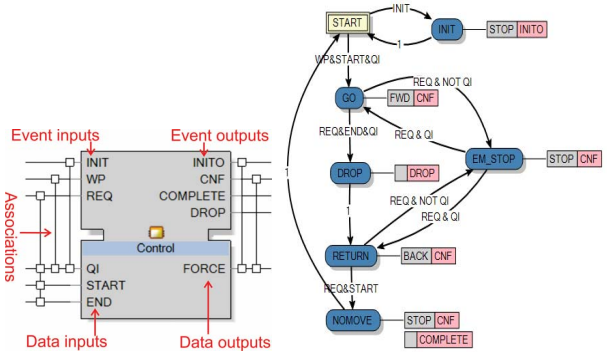


Fig. 4: The interface and ECC of a basic function block.

ECCs are Moore-type finite state machines that describe the execution behaviors of basic function blocks. The ECC of the Control block in Fig 4 consists of 6 states with START as the initial state. Each state is associated with a finite sequence of *actions* (algorithm executions and output events emissions). E.g., for state INIT, the actions include algorithm *STOP* and the output INIT0.

The outgoing transitions of any state in the ECC may have a *guard*. A guard is an input event, a Boolean expression over the variables of the block, an input event-Boolean expression combination, or 1 (always-enabled). A transition is enabled when its guard evaluates to true. As an example, consider the transition START $-^{\text{WP\&START\&QI}}\rightarrow$ GO in Fig. 4. This transition fires when the input event WP appears *and* the expression START & QI is true. Similarly, transitions with the guard 1 are always enabled. Transitions of each state in the ECC are *ordered* by the user, so in the case when two or more transitions are enabled, only the highest priority transition fires, resulting in deterministic execution.

Other than basic blocks, IEC 61499 programs may also contain *service-interface*, and *composite* blocks. Service-interface function blocks provide standard input/output and network interfaces. E.g., *PUBLISH* and *SUBSCRIBE* blocks allow TCP/IP or UDP/IP communications between a system and its environment.

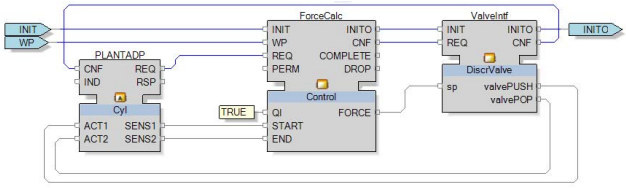


Fig. 5: Composite function block.

A *composite* function block contains a finite network of function block instances. The composite block in Fig. 5 shows a closed-loop connection between a controller (ForceCalc), a plant interface (PLANTADP), and a valve interface (ValveIntf).

An IEC 61499 *application* is a complex composite block implementing a specific task of a system. Fig. 6 presents a control application for the single cylinder system. Once a workpiece is placed in front of the pusher (that is detected by

WPS sensor), the desired service of this system is to push the workpiece to the destination sink and retract the pusher to the initial state. In the function block application, the instance WP1H simulates a button which simulates the arrival of a work-piece. The controller of the cylinder CTL1 (an instance of composite FB `CControlA`) is connected in closed-loop with the cylinder (represented as FB Plant1). The application in Fig. 6, contains two event connections and one adapter connection. E.g., the output event IND of WP1H connects to the input event WP of CTL1. The adapter connection CTL1.PLANTADP → Plant1.CTL encapsulates eight connections in both directions to implement closed-loop plant-controller interaction.

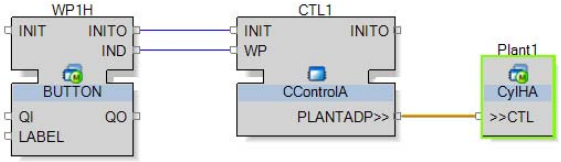


Fig. 6: Controller application for a single cylinder system.

The ECC of the block Control, shown in Fig. 4, implements the cylinder control logic as follows. From its initial state INIT, a transition to state INIT fires when the input event INIT is read. In state INIT, the initializing algorithm STOP executes and the output event INITO is emitted. Once initialized and having returned to state START via the always-enabled transition, the ECC moves to state GO and executes the algorithm *FWD* when the corresponding transition WP& START & QI evaluates to TRUE. This algorithm changes the value of the output variable FORCE, and the emission of the associated event CNF makes this command available to block ValveIntf (connected to the controller block as shown in Fig. 5). When the cylinder has fully extended, a transition to state RETURN triggers to retract and then reinitialize the cylinder.

B. Case study implementation

An application implementing control of the two-pusher system from Fig. 1 (a) is shown in Fig. 7. Along with the basic functionality of each pusher, this application implements mutually exclusive access to the overlapping areas of operation using a ring token protocol.

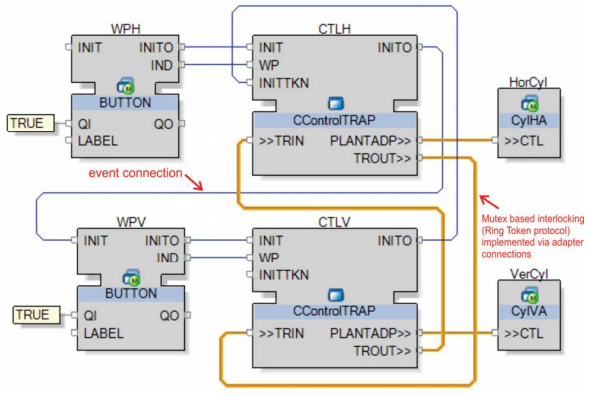


Fig. 7: Top level function block application implementing control of two orthogonal pushers with interlocking.

As one can observe, the interface of controller function blocks CControlTRAP has been extended from CControlA by one plug and one socket. This extension is to implement the ring token protocol of mutually exclusive access of a cylinder to the operation area. The internals of CControlTRAP are presented in Fig. 8.

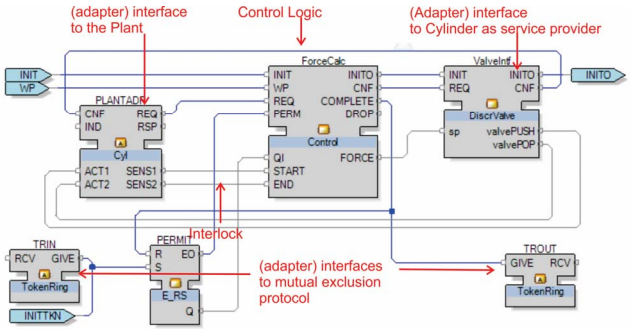


Fig. 8: Function block network of `CControlTRAP`.

The interface of Control function block has been extended with the PERM event input that is used to model the receipt of token in the ring token protocol. The permission to operate is determined by the Boolean input QI that is set by a preceding member of the ring. The state machine has been also modified in order to implement token receipt and return as shown in Fig. 9. Token is immediately returned to the next ring member unless workpiece is present, so that state machine is in WAITTKN state.

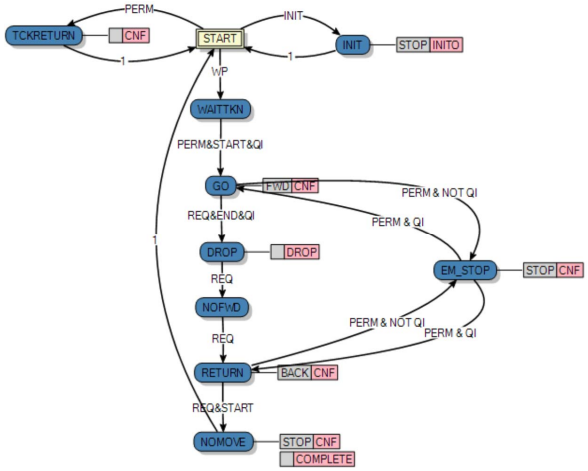


Fig. 9. State machine of the modified controller

B. Executing Distributed IEC61499 Programs

The design process in IEC 61499 consists of two steps. In the first, an application is created as a hardware independent executable specification of a distributed system's functionality. In the second step, it is further refined by including a particular distributed system architecture that is specified by means of devices connected by communication network segments. Function blocks of an IEC61499 application are mapped at this stage to *devices*. Each device

can be internally structured from resources that represent abstractly a platform that can schedule and execute function blocks mapped to it. A device represents an independent physical platform that can execute multiple resources (e.g. a PLC). When an application is distributed amongst different resources, cross-resource communication between the distributed parts is carried out using network connections.

Most IEC61499 development environments allow users to distribute applications by mapping parts to different resources, and then automatically insert the relevant network connections (in the form of service-interface blocks) to carry out the inter-resource communication. Each resource typically uses a run-time environment (a kind of virtual machine) to schedule and execute function blocks, and the device is responsible for carrying out the translation of network interactions to events and variables to be read or written by the function blocks executing in its resources.

IV. COMPARISON

After implementing the simple two-cylinder example in SystemJ and IEC61499, we make the following observations:

- ***Structure**:* SystemJ and IEC 61499 use reactions and basic blocks executing ECCs, respectively, constructs that are based on finite state machines. Both allow building more complex behaviors by connecting multiple reactions or basic blocks into clock-domains/systems or composite blocks respectively. This compositional structure serves well for distributed systems in general, and both these languages provide solid support for this.
- ***Component Reuse**:* The programs in Fig. 2 and Fig. 5 reuse a single-cylinder controller to control the two cylinders of the case study. In general, both languages allow defining behaviors (reactions, clock domains, systems, basic blocks, composite blocks, etc.) and then declaring instances of these behaviors to create complex programs from component reuse.
- ***Mapping**:* For programs written in both languages, users must *connect* or map I/O between interacting behaviors. Mapping allows users to reuse components deploy programs over different hardware configurations.

Both languages also differ in the following manner:

- ***Hierarchy**:* SystemJ allows forking of new reactions within a reaction to any depth, thus effectively implementing behavioral hierarchy of reactions. Finite nesting of abort [12] statements allows preemptions with different priorities. Such hierarchy is not possible in IEC61499 function blocks. Even composite function blocks allow only *structural* hierarchy, and modeling Statecharts-like behaviors requires significant effort.
- ***Flexibility in communication mappings**:* In SystemJ, signals shared between the reactions of the same clock domain are mapped automatically and by name. SystemJ also has a default mapping of channels if the program runs on the same JVM: channels are implemented in shared memory and hence no mapping is required. Other channels and interface signals must be mapped by users via a configuration file (see section 2) via shared memory or networking-based communications. In IEC61499, mapping of function block inputs and outputs to process and communication interface signals is done after the function blocks of an application are mapped (allocated) to particular devices. The devices contain so called service interface function blocks that represent physical signals. Consequently, IEC61499 function block instances can be reused in different ways by changing the event and/or variable connection mappings. SystemJ allows users to provide different mappings too, except for signal mappings between reactions in the same clock domain. The additional flexibility in IEC61499 requires extra mapping effort, whereas mapping (of signals) by name in SystemJ is automatic. This difference can be explained by the strong encapsulation concept of IEC 61499 functions blocks which have been conceived as distributed programming constructs. Due to this, only message passing and no global variables can be used for inter-component communication.
- ***Mapping stage**:* In SystemJ, the user must have a clear idea about the synchronous and asynchronous behaviors of a program before coding them. Reactions can be defined and instantiated within any clock domain, but the clock domain is a minimal entity that is compiled independently and then can run on any specified execution platform (resource) specified in the configuration file of the program, where the details of execution platforms and used communication mechanisms are described. Having prior knowledge about mapping allows designers to write efficient and compact code for SystemJ. In IEC 61499, since there is such distinction between execution semantics at any level, we can write programs without prior knowledge of which blocks execute together (say on a single resource). The IEC61499 programs can be re-configured easily to run on different resource-device combinations. This difference also highlights the domain-specific design metrics for the two languages; in SystemJ, efficiency and compactness of code in embedded systems are of high priority, while in IEC61499, re-configurability is more important.
- ***Execution semantics**:* SystemJ has well-defined delayed synchronous semantics. The semantics removes ambiguities and allows designers to write correct-by-construction code with proper knowledge of how it executes. Although IEC61499 does not define execution semantics formally, it is rather message-passing semantics, where ECCs of basic blocks react only to input events by emitting outputs and (probably) changing their state. However, ensuring that events are fully ordered across a distributed architecture and that blocks can react to all event sequences requires substantial effort. While SystemJ-like delayed synchronous semantics for IEC 61499 was proposed in [14], it does not allow rendezvous-style communications between distributed programs.

- ***Communication constructs**:* SystemJ supports two well-defined communication constructs: signals and channels. Both constructs have different uses, and may bind to values (primitives or objects) based on definition. On the other hand, IEC61499 provides events as the only communication construct. Variables and event-variable associations are used to additionally bind events to values. It is therefore possible that an output event of a block updates specific output variables of that block, while a connected input event of another block may cause the sampling of a completely different set of variables. This is more flexible but also more dangerous than SystemJ.
- ***Coding style**:* While SystemJ programming is primarily textual, IEC61499 uses visual artifacts with textual coding limited only to writing algorithms and state and transition labels in ECCs. This difference indicates the domain-specific preferences for the two languages. In embedded systems, the focus has always been on textual coding such that programmers can exercise minute control over the code. On the other hand, IEC61499 continues in the vein of visual IEC61131 languages such as ladder logic and sequential function charts, where the focus has been on ease of specification and automatic code generation. There is a growing popularity of visual methods in embedded systems, including such examples of block-diagram languages as Matlab, LabView, Scade, Rubus, etc. None of these, however, aims at distributed systems development, giving IEC 61499 a clear advantage in this area of application.

V. CONCLUSIONS

The dream of an ideal programming language fades rapidly when domain-specific differences between existing languages surface. Studying SystemJ and IEC 61499 leads to a similar conclusion: both languages are optimized to their domains which justify their inherent differences. However, when we went beyond the veil of domain-specific features, we uncovered similarities between the two languages. We found that some strengths of one language can be readily assimilated into and enrich the other. E.g., SystemJ can benefit from adopting visual elements of IEC 61499, while IEC 61499 can benefit from clarifying its execution semantics. We can also lower user effort in IEC61499 coding by allowing default mappings (by name as in SystemJ) and hierarchical behaviors via Statecharts-based [20] extensions to ECCs. Similarly, SystemJ can gain flexibility by allowing dynamic mapping of reactions to clock domains.

Future work in this area includes developing and analyzing languages on two levels: domain-specific and domain-independent, and to develop a minimal domain-independent language that can be extended with domain-specific details for use in different areas of distributed computing.

REFERENCES

[1] J. A. Feldman, J. R. Low, and P. Rovner, "Programming distributed systems," in *Proceedings of the 1978 annual conference*. ACM, 1978, pp. 310–316.

[2] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *ACM Sigplan Notices*, vol. 38, no. 5. ACM, 2003, pp. 1–11.

[3] W. C. Rounds and H. Song, "The o-calculus:¨ A language for distributed control of reconfigurable embedded systems," in *Hybrid Systems: Computation and Control*. Springer, 2003, pp. 435–449.

[4] P. Leitao,˜ "Agent-based distributed manufacturing control: A stateof-the-art survey," *Engineering Applications of Artificial Intelligence*, vol. 22, no. 7, pp. 979–991, 2009.

[5] F. Gruian, P. Roop, Z. Salcic, and I. Radojevic, "The SystemJ approach to system-level design," in Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE, 2006, pp. 149–158.

[6] Function Blocks, International Standard IEC 61499, International Electrotechnical Commission, Geneva, Switzerland, Second Edition, 2012

[7] V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State of the Art Review", *IEEE Transactions on Industrial Informatics*, 7(4), 2011, pp. 768-781

[8] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, no. 12, pp. 1523–1543, 2000.

[9] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Advanced Research in Asynchronous Circuits and Systems, 2000.(ASYNC 2000) Proceedings. Sixth International Symposium on*. IEEE, 2000, pp. 52–59.

[10] Programmable Controller - Part 3: Programming Languages, International standard IEC 61131-3. Geneva: International Electrotechnical Commission, 1993

[11] N. A. Lynch, *Distributed algorithms*. Morgan Kaufmann, 1996.

[12] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," *Computer Languages, Systems & Structures*, vol. 36, no. 4, pp. 317–344, 2010.

[13] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The Synchronous Languages 12 Years Later," in *Proceedings of the IEEE*, 2002, pp. 64–83.

[14] L. Yoong, P. Roop, V. Vyatkin, and Z. Salcic, "Synchronous Execution of IEC 61499 Function Blocks Using Esterel," in *Proceedings of IEEE International Conference on Industrial Informatics 2007*, 2007, pp. 1189–1194.

[15] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[16] M. Schoeberl, "A java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1, pp. 265–286, 2008.

[17] R. B. Smith, "Spotworld and the sun spot," in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*. IEEE, 2007, pp. 565–566.

[18] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java on the bare metal of wireless sensor devices: the squawk java virtual machine," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 78–88.

[19] U. D. Atmojo, Z. Salcic, and I. K. Wang, "System-level approach to the design of collaborative distributed systems based on wireless sensor and actuator networks," in *Proceedings of 5th International Workshop on Smart Environments and Ambient Intelligence, SENAmi*, 2013.

[20] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, June 1987. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(87)90035-9