Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 53(3), 294–310(2012)

*Siniša Srbljić, Dejan Škvorc, Miroslav Popović*

# Programming Languages for End-User Personalization of Cyber-Physical Systems

DOI 10.7305/automatika.53-3.84
UDK 004.43.056
IFAC 2.8.3

**Original scientific paper**

The increased usage of smart devices and appliances opens new venues to build applications that integrate physical and virtual world into consumer-oriented context-sensitive cyber-physical systems (CPS). Since physical processes are dynamic, concurrent, event-driven, and powered by various sensors, controllers, and actuators, a combination of *service-oriented architecture* (SOA) and *event-driven architecture* (EDA) is the most promising software architecture for virtualization of heterogeneous components into interoperable application building blocks. In this paper, we propose a CPS design paradigm where devices, such as sensors, controllers, and actuators, are virtualized into *environmental services*. To support event-driven workflow coordination, we designed special-purpose *coopetition services* that provide fundamental EDA characteristics, such as decoupled interactions, many-to-many communication, publish/subscribe messaging, event triggering, and asynchronous operations. Based on these two groups of services, we present a design of event-driven service composition languages that target two distinct groups of developers. Using Python as an example, we present a transformation of arbitrary general-purpose programming language into an event-driven service composition language for developers familiar with parallel programming using operating system kernel mechanisms. On the other hand, we present the design and cognitive evaluation of an end-user language, whose 2D tabular workspace resembles the process of sketching an automation application on a sheet of paper.

**Key words:** Cyber-physical systems, Service-oriented event-driven programming, Multi-device applications, Tabular programming

**Krajnjem korisniku prilagođeni programski jezici za poosobljavanje računalom upravljanih okolina.** Povećanom uporabom suvremenih elektroničkih uređaja otvaraju se nove mogućnosti za izgradnju primjenskih programa koji objedinjuju fizički prostor i informacijske sustave u korisniku usmjerene računalom upravljane okoline. Suvremeni prostori opremljeni su različitim vrstama osjetila, upravljača i pokretačkih uređaja koji vremenski usklađeno upravljaju dinamičkim i događajima poticanim paralelnim procesima. Spregom *uslužno usmjerene* i *događajima poticane arhitekture* omogućen je pristup raznorodnim fizičkim uređajima u obliku međusobno sukladnih gradivnih komponenti primjenskih programa. U radu je predložena paradigma izgradnje računalom upravljanih okolina u kojoj se uređajima iz okoline pristupa putem *programskih usluga*. Za potrebe oblikovanja događajima poticanih tijekova izvođenja programa, oblikovan je poseban skup *usluga suradnje i natjecanja*. Te usluge ostvaruju osnovne značajke arhitekture zasnovane na događajima, kao što su neizravno međudjelovanje, komunikacija u grupi, objavi/pretplati komunikacija, pokretanje događaja i asinkrone operacije. Na osnovi tih dviju skupina usluga, oblikovana su dva jezika za događajima poticanu kompoziciju usluga. Na primjeru jezika Python, prikazano je preoblikovanje jezika opće namjene u jezik za događajima poticanu kompoziciju usluga namijenjen razvijateljima paralelnih programa primjenom mehanizama jezgre operacijskog sustava. S druge strane, prikazano je oblikovanje i kognitivno vrednovanje tabličnog jezika namijenjenog krajnjem korisniku, gdje oblikovanje primjenskog programa unutar dvodimenzionalne radne plohe nalikuje skiciranju međudjelovanja skupine uređaja na listu papira.

**Ključne riječi:** računalom upravljane okoline, programiranje zasnovano na događajima poticanoj kompoziciji usluga, primjenski programi za upravljanje radom skupine uređaja, tablično programiranje

## 1 INTRODUCTION

Present-day living and working environments are equipped with various types of devices for surveillance, ambient intelligence, traffic and transportation control, industrial automation, elderly people care, or health care. These devices morph successfully into our physical living

and working space through *cyber-physical systems* (CPS) that integrate physical world of devices with virtual world of information technology [1, 2].

Cyber-physical systems are integrations of computation and physical processes. Figure 1 presents elements of a CPS that consists of the *device space* and *CPS application design space*. The *device space* is a physical space where CPS application is executed and where interactions of CPS with physical environment occur. The *CPS application design space* is a virtual space where computational processes that drive the operation of CPS is designed and implemented.

The *device space* includes a wide range of sensors, controllers, and actuators interconnected through communication networks [3, 4]. Furthermore, the device space includes *computer network* that provides the execution environment for computational part of CPS that control the operation of devices. Devices are accessible from computer network through *communication network*.

The *CPS application design space* provides the APIs to access the device functionalities and programming languages for composing devices into CPS applications. To be suitable for integration into CPS applications, devices have their representations in CPS application design space.

Openness to physical environments on one side, and people that live and work in these environments on the other, requires online, open-ended, interactive, concurrent, and event-driven information systems that drive the operation of CPS. As a result, the most suitable technology for development of such information systems is based on combination of *service-oriented architecture* (SOA)

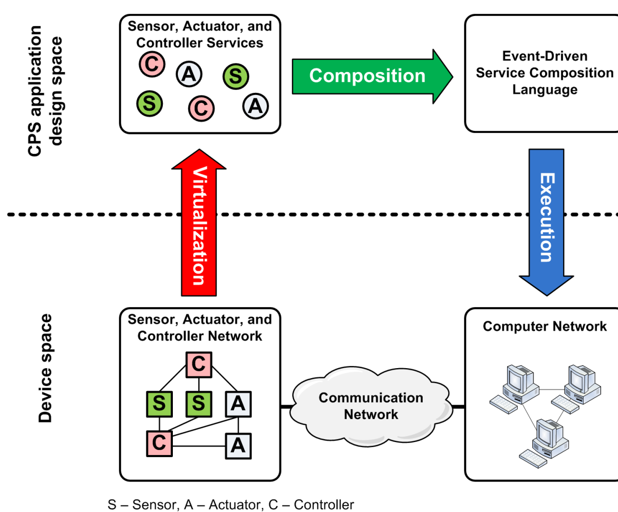


*Fig. 1. Cyber-physical system based on event-driven service composition*

and *event-driven architecture* (EDA) [2, 5-10]. In this paper, we present a design of programming languages for development of CPS applications based on combination of SOA and EDA. To build a CPS application, programmer defines a control logic that coordinates the operation of a set of devices. We propose an event-driven SOA where devices, such as sensors, controllers, and actuators, are exposed to application developers as web services. To handle events in distributed environment, we developed special-purpose event-handling services.

In an environment where embedded devices and event-handling mechanisms are virtualized as services, *service composition* is used as a design paradigm to connect mutually independent services into domain-specific CPS [11]. *Service composition languages* are, therefore, used as process description languages to define control processes from which the execution of devices is orchestrated. We present two programming languages for two different groups of application developers with different domain knowledge and programming skills. Using Python as an example, we present a transformation of arbitrary general-purpose programming language into an event-driven service composition language for developers familiar with parallel programming using operating system kernel mechanisms. On the other hand, we present the design and cognitive evaluation of an end-user language, whose 2D tabular workspace resembles the process of cognitively comprehensible sketching a multi-device CPS application on a sheet of paper.

The rest of the paper is organized as follows. In Section 2, we discuss the CPS design challenges that impact the design of programming languages for implementation of CPS control processes. In Section 3, a generic example of CPS application that is used throughout the paper is given. In Section 4, we propose a combination of EDA and SOA as a fundamental software architecture upon which CPS applications are built. In Section 5, we describe a language design methodology where event-driven service composition languages are derived from widely accepted scripting and spreadsheet languages. In Section 6, service composition language *PIEthon*, which is SOA&EDA-ready version of Python, is described. Section 7 presents an end-user tabular language *HUSKY*. Section 8 concludes the paper.

## 2 LANGUAGE DESIGN CHALLENGES

Since CPS are integrations of physical and computational elements whose operation has visible and tangible impact on human surroundings or even life, there are several challenges that have to be met while designing such systems. These challenges also

impact the design of programming languages used for implementation of such systems, since some CPS properties have to be directly expressed in the language used by system developer.

**Networking.** Opposite to traditional embedded computer systems that operate as closed boxes that do not expose their functionality to the outside, modern CPS will benefit from being networked. Networking of domain-specific CPSs that usually operate in isolation into multi-domain CPS raises two questions. First, how to enable global networking of CPS components that belong to different administrative domains, either being connected permanently into long-lived applications or dynamically into on-demand applications. Second, how to abstract technologically diverse CPS components into unified application modeling space and expose them to developers as uniform application building blocks.

**Timing.** Reliable and safe execution of physical processes often assumes completion of given operation within a specific time frame. Programming abstractions CPS designers use to develop CPS control processes have to provide the ability to express timing constraints in user applications.

**Event-driveness.** Computational parts of CPS are naturally event-driven processes triggered by events from physical components. Most of today's embedded systems react to sensor stimuli from physical environment, process the stimuli, and control the operation of actuators to react back to the environment. Events are occurring in various forms, from simple notification signals to data messages to complex structures that require multi-stage downstream processing. Programming abstractions used for implementation of CPS have to provide the ability to handle events in user programs and to control the flow of events through the system.

**Concurrency.** An intrinsic property of real-world physical processes is concurrency. Composing multiple physical processes into networked and coordinated CPS requires programming abstractions to explicitly express concurrent composition of CPS segments.

**End-user orientation.** As CPS go beyond their traditional domains, such as industrial automation, traffic control, or medical instrumentation, and become an integral part of human living and working environments, such as homes, offices, playgrounds, and shopping malls, they have strong impact on users' privacy and quality of experience. Since full commodity of living in such environments depends on individual user preferences, one of the key challenges in CPS design is to enable end users to tailor CPS behavior towards their personal needs, expectations, and habits. For example, interpretation of sensor stimuli from wind meter may require different processing in an application used by meteorologist than in one used by farmer, sailor, or alpinist. Users of different profiles need application design workspaces and modeling languages tailored to their needs, knowledge, and skill sets.

## 3 CPS APPLICATION EXAMPLE

CPS-driven smart environments are complex real-time systems. For example, automating an apartment building involves development of a sophisticated control system for each floor. The system monitors and analyzes state, and controls operation of numerous home appliances spread across the building. Therefore, these systems are based on a large number of control patterns with complex interrelationships. The overall logic of the system manages a large number of sensors and actuators interconnected through *sensor-controller-actuator* patterns. Controllers process the data given by sensors and prepare the control inputs for actuators. Figure 2 presents an example of CPS based on four sensor-controller-actuator patterns. We keep the example simple and general, but useful enough to present the main features of our programming languages and show how we deal with sophisticated event-driven control processes. The initiation *Sensor0-Controller0* pattern triggers *Sensor1-Controller1-Actuator1* and *Sensor2-Controller2-Actuator2* patterns, which in turn trigger the finalization *Controller3-Actuator3* pattern.

The simplest implementation architecture for the described CPS is based on a single-task application. The single-task application includes sequence of statements for execution of control patterns. Control patterns access the sensors, actuators, and controllers through memory devices that store their data. The main advantage of single-task application is its simplicity. However, this type of architecture is not suitable for implementation of event-driven automation processes. In such processes, there is large number of distributed and embedded devices. These devices act independently and trigger various events.

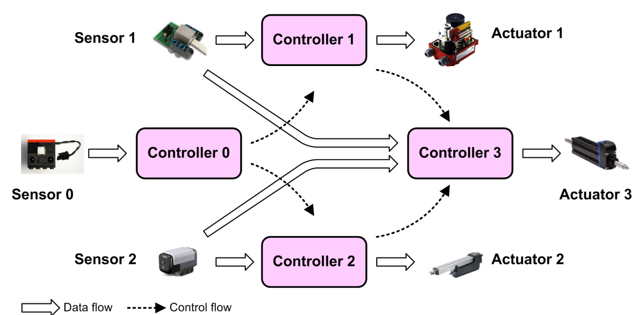


*Fig. 2. An example of CPS based on concurrent control loops*

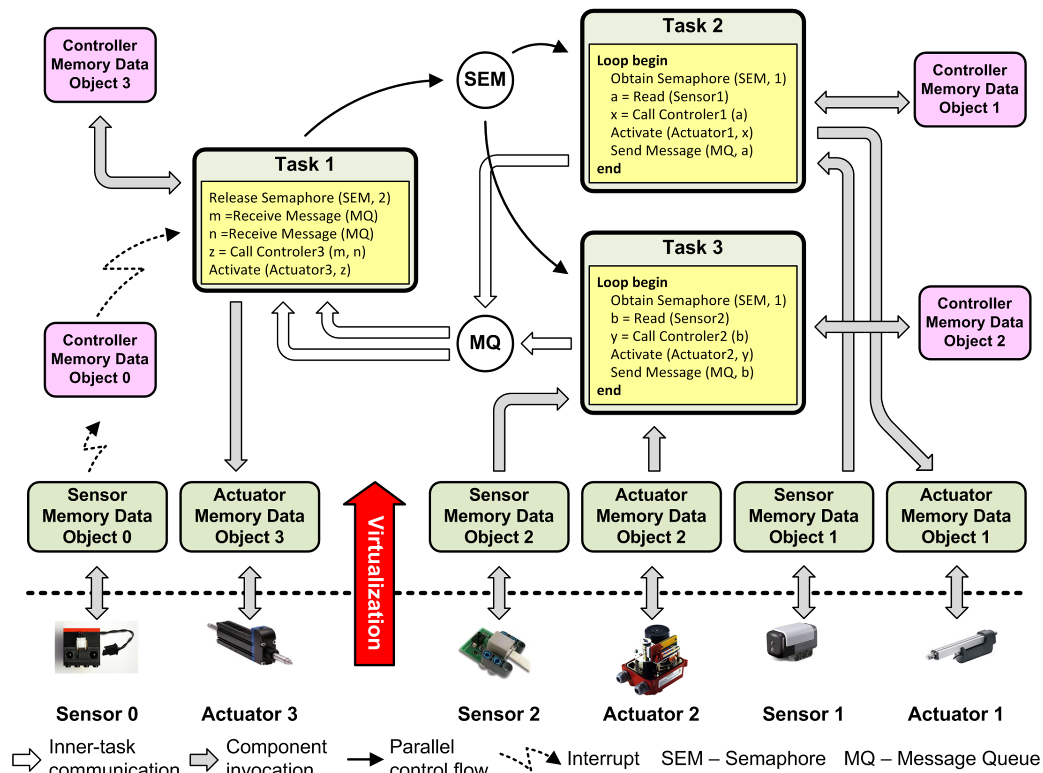


*Fig. 3. Implementation of CPS based on multitasking architecture*

These events must be processed using different control patterns and multiple events may have to be processed at the same time and within given time frame.

To enable processing of events in a concurrent and distributed environment, automation processes are usually implemented using multitasking architecture. Environment automation process logic is implemented using a set of tasks. Tasks execute control patterns and coordinate their execution using a set of specialized collaboration mechanisms, such as *semaphore* and *message queue*, for task synchronization and asynchronous communication.

Multitasking architecture enables processing of events using different types of control patterns with different levels of complexity. Tasks may gather and process events independently, concurrently, or in any other order compliant with environmental automation process requirements. These features make the architecture flexible and suitable for the implementation of a wide range of automation processes.

Figure 3 presents an implementation of the event-driven CPS control process shown in Fig. 2 using a multitasking architecture. The process automation logic is implemented using three tasks collaborating using *semaphore* (*SEM*) and *message queue* (*MQ*) mechanisms. The logic of tasks shown in Fig. 3 is expressed in a pseudo code resembling the constructs of a typical scripting language.

*Task 1* implements the initiation *Sensor0-Controller0* loop and the finalization *Controller3-Actuator3* loop. *Task 2* implements the *Sensor1-Controller1-Actuator1* loop. *Task 3* implements the *Sensor2-Controller2-Actuator2* loop. After being triggered by the *Sensor0* trough the *Controler0*, *Task 1* triggers the execution of *Task 2* and *Task 3* by releasing the semaphore *SEM* (*Release Semaphore(SEM, 2)*). Until being triggered by *Task 1*, *Task 2* and *Task 3* stay blocked on the semaphore *SEM* (*Obtain Semaphore(SEM, 1)*). After they finish the execution of control loops by reading the data from sensors (*Read(SensorX)*), preparing the control data (*Call(ControllerX, Y)*), and activation of actuators (*Activate(ActuatorX, Y)*), *Task 2* and *Task 3* send the results back to *Task 1* through the message *MQ* (*Send Message(MQ, Y)*). Upon receiving two messages from the message queue *MQ* (*Receive Message(MQ)*), *Task 1* continues its execution with the finalization *Controller3-Actuator3 loop*.

## 4 SOA&EDA-BASED CPS

Design of a programming language for development of CPS applications begins with a decision in what form environmental devices appear in the language and how programmers manipulate with them. Many different technologies and out-of-the-box solutions for *virtualization* of devices as programming elements exist on the market. Most of these products are specialized in some specific domain, such as surveillance, energy consumption optimization, or health care. Solutions are usually mutually incompatible and use nonstandard communication protocols, data encoding formats, and automation process description languages.

There are several initiatives that are developing open, common, network-independent communication interfaces for connecting sensors, actuators, and controllers. For example, the key feature of IEEE 1451 standard is the definition of Transducer Electronic Data Sheets [12] (TEDS, http://ieee1451.nist.gov). The TEDSs are memory devices that store sensor, actuator, and controller related information. The goal of this standard is to allow the access to data stored in TEDSs that are attached to sensors, actuators, and controllers through a common set of interfaces. Thus, sensors, actuators, and controllers become indirectly accessible from a computer network through devices like TEDSs.

In the last couple of years, *Web Services* [13] and *service-oriented architecture* (SOA) became the most widely accepted technology for development of applications comprising of heterogeneous components. *Web Services* enable virtualization of heterogeneous physical devices as services in CPS application design space. For example, the *EUREKA ITEA software Cluster SODA project* [14] has created a service-oriented ecosystem for high-level communications between computer systems and smart embedded software components in low-cost devices in the so-called "*web of objects*". This enables all types of devices to communicate and interact using the same language. Uses for this technological innovation will be in a wide range of applications for industrial automation, automotive electronics, home and building automation, telecommunications, and medical instrumentation. The concept was adopted globally by OASIS as the *Devices Profile for Web Services* (DPWS) standard in mid 2009 [15].

To support global networking of distributed devices and enable their technology-transparent composition, we propose to use SOA as basic software architecture upon which CPS applications are built. However, fundamental property of any application that integrates devices such as sensors and actuators is event-driven execution. Therefore, the architecture and programming language for development of such applications should enable seamless integration of these devices into event-driven workflows. Since basic SOA does not support event-driven workflows, it is augmented with special-purpose event-handling services [16-19]. These services implement fundamental characteristics of *event-driven architecture* (EDA), such as decoupled interactions, many-to-many communication, publish/subscribe messaging, event triggering, and asynchronous operations. In [19], we proposed an event-driven service-oriented architecture for development of SOA&EDA applications. The architecture is shown in Fig. 4. It is based on three types of components: *application-specific services* for application-specific computations, *coopetition services* for handling events in distributed environment, and user programs that connect these services into *event-driven service composition*.

When applied for development of CPS, application-specific services become *environmental services* through which devices embedded into the environment are virtualized into automation application. *Coopetition services* are used for inter-task communication, synchronization, and event triggering, as sketched in Fig. 3. We have developed three types of *coopetition services*: *TokenCenter* for synchronization of user tasks, *Queue* for decoupled asynchronous communication, and *BrokerCenter* for publish/subscribe messaging [16-19]. Finally, user programs written in event-driven service composition language contain the logic for coordination of environmental services that operate on devices and handling of events through *coopetition services*.

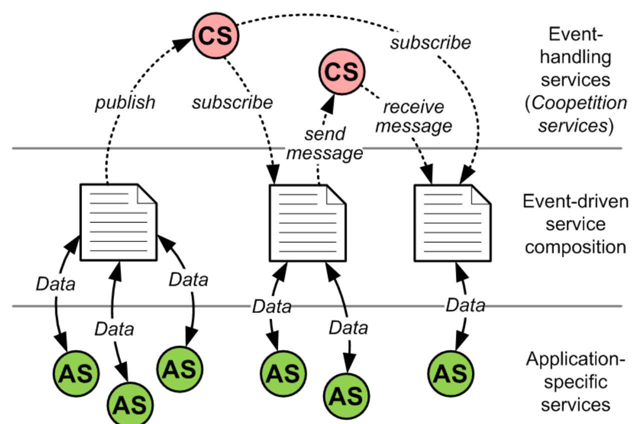


*Fig. 4. Event-driven SOA based on application-specific and event-handling services*

## 5 LANGUAGE DESIGN METHODOLOGY

To build a CPS application, programmer defines a control logic that coordinates the operation of a set of devices. If these devices are virtualized into web services, then *service composition* is used as a CPS application design paradigm. A language used for implementation of SOA&EDA based CPS applications is considered SOA-ready if it contains first-class *primitives for invocation of environmental services*. Furthermore, since the complexity of event processing is hidden behind the coopetition services, a language is considered EDA-ready if it contains *primitives for invocation of coopetition services*.

In our previous paper [19], we discussed the design of event-driven service composition languages that were primarily designed for implementation of business applications. These languages target developers already familiar with standardized languages for development of SOA-based applications, such as WS-BPEL [22, 23] and similar XML-based languages. We proposed the language extensions that augment WS-BPEL with EDA properties.

In this paper, we discuss the design of event-driven service composition languages that target practitioners in environmental automation domain, where the adoption of WS-BPEL and XML-based languages is not a common practice. On the other side, to satisfy end-users' needs to personalize the behavior of CPS to their individual habits and expectations, we discuss the design of event-driven service composition languages for users not formally trained in programming. Our languages for implementation of CPS applications are based on scripting languages and spreadsheets. The influential languages that drove the design of proposed language suite are presented in Fig. 5.

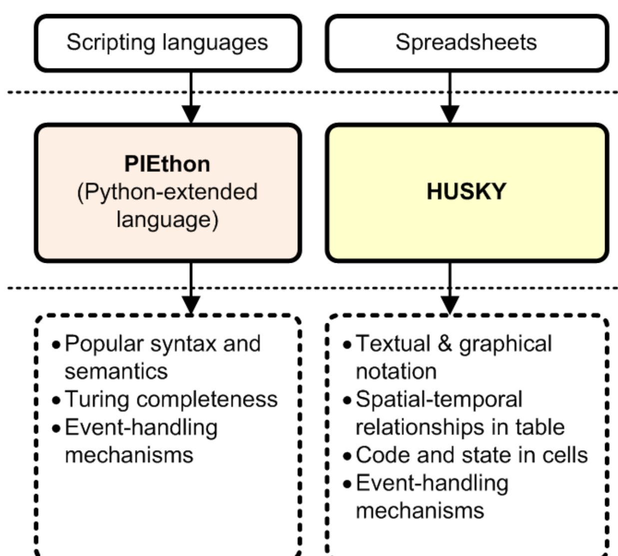


*Fig. 5. Design of event-driven service composition automation languages*

Since scripting languages, such as Python and Perl, are widely used general-purpose languages for development of component-based applications, we developed an event-driven service composition automation language derived from Python. We chose Python due to its popularity and wide acceptance by a wide and open user community. To be SOA-ready, we developed a *Python module for invocation of Web Services* from Python programs. To be EDA-ready, a *module for invocation of event-handling services* is developed. New language is called *PIEthon* (*Programmable Internet Environment Python*). In similar way as we extended Python into *PIEthon*, any general-purpose system or scripting language can be extended into an event-driven service composition language.

To enable a comprehensive user-friendly representation and management of CPS control logic, we developed a spreadsheet-like tabular language named *HUSKY*. Because of two-dimensional design workspace, *HUSKY* programs resemble the CPS control process sketched as a graphical scheme on a sheet of paper. *HUSKY* is a semi-graphical programming language that combines properties of textual and graphical languages.

## 6 PIETHON: SOA&EDA-READY PYTHON

Specialized service composition languages, such as SSCL [19-21], enable rapid development of event-driven SOA applications due to compact and domain-specific set of language elements. However, designing a new language imposes a learning curve on prospective developers since they have to spend time to learn the lexical, syntax, and semantic features of the language. Our goal was to design an automation language for SOA&EDA-based applications that is not tightly coupled to SOA community, but rather familiar to wide population of software developers. Second, automation logic often requires computational logic which is not available in any service used in composition. For example, if data formats of two services that should be linked together do not mach, we need a computational logic for data format conversion. Such scenarios require Turing complete programming language for service composition, a feature which is not available in coordination languages such as SSCL.

To minimize the learning curve and achieve the Turing completeness of automation language, we propose to reuse the features and expressiveness of a general-purpose programming language. Scripting languages are nowadays taking a leading role in component software design,

*Interrupt Handler Task*

```
01: while True:
02:     C = ServiceLib.Execute("http://mynet.com/sensor0", "WaitForInterrupt")
03:     BrokerCentLib.Publish("http://mynet.com/BrokerCenter", C)
```

*Task 1*

```
01: BrokerCentLib.Subscribe("http://mynet.com/BrokerCenter", Terms, "http://mynet.com/Controller0")
02: TokenCentLib.Release("http://mynet.com/TokenCenter", 2)
03: M = QueueLib.Receive("http://mynet.com/Queue")
04: N = QueueLib.Receive("http://mynet.com/Queue")
05: Z = ServiceLib.Execute("http://mynet.com/controller3", "Calculate", M, N)
06: ServiceLib.Execute("http://mynet.com/actuator3", "Output", Z)
```

*Task 2*

```
01: while True:
02:     TokenCentLib.Obtain(http://mynet.com/TokenCenter, 1)
03:     A = ServiceLib.Execute("http://mynet.com/sensor1", "Read")
04:     X = ServiceLib.Execute("http://mynet.com/controller1", "Calculate", A)
05:     ServiceLib.Execute("http://mynet.com/actuator1", "Output", X)
06:     QueueLib.Send("http://mynet.com/Queue/E8", A)
```

*Task 3*

```
01: while True:
02:     TokenCentLib.Obtain(http://mynet.com/TokenCenter, 1)
03:     B = ServiceLib.Execute("http://mynet.com/sensor2", "Input")
04:     Y = ServiceLib.Execute("http://mynet.com/controller2", "Calculate", B)
05:     ServiceLib.Execute("http://mynet.com/actuator2", "Output", Y)
06:     QueueLib.Send("http://mynet.com/Queue", B)
```

*Fig. 6. Multitasking implementation of CPS control process in PIEthon*

gathering programmers into the largest programmer community. To open CPS development to this programmer community, we decided to upgrade the scripting languages to be SOA and EDA-ready languages. As an example in this paper, we are using Python. Since standard Python does not include built-in statements for service invocation, we developed Python modules for managing *Web Services* invocations and handling events through *coopetition services*. The package including Python augmented with service invocation and event handling modules is called *PIEthon* (*Programmable Internet Environment Python*). *PIEthon* is a SOA&EDA-ready version of Python.

Development of CPS applications in *PIEthon* is similar to multitasking or multithreaded parallel programming where set of tasks mutually collaborate using operating system kernel mechanisms, such as semaphore, message queue, or pipe [24]. *PIEthon* is, therefore, intended for CPS developers familiar with operating system level programming.

When *PIEthon* is used, the CPS application shown in Fig. 3 is implemented using four tasks: *Interrupt Handler Task* dispatches interrupts from *Sensor0* to *Task 1*, while *Tasks 1 - 3* implement the rest of the automation process. The tasks are implemented using the code presented in Fig. 6. To invoke an environmental service, we use a generic service invocation method *ServiceLib.Execute*. The method accepts the service location, the operation name, and the operation arguments as parameters.

To handle events, we use three *coopetition services*: *Queue*, *TokenCenter*, and *BrokerCenter*. Presented *PIEthon* code shows an example of using a *Queue* where *Task 2* and *Task 3* send values to *Task 1*. The communication consists of two methods: a *QueueLib.Send* method to add a message to the message queue, and a *QueueLib.Receive* method to acquire a message from the queue. Both methods accept the message queue service location as a parameter, while *QueueLib.Send* method accepts the message to be queued as an additional parameter.

Presented *PIEthon* code also shows an example of synchronization of concurrent tasks. In this example, the *Task 1* runs before the *Task 2* and *Task 3*. Synchronization consists of two methods: *TokenCentLib.Obtain* and *TokenCentLib.Release*. *TokenCentLib.Obtain* method acquires tokens from the *TokenCenter*. The parameter specifies the *TokenCenter* service location and the number of tokens to be acquired. Method *TokenCentLib.Release* returns tokens to the *TokenCenter*. The parameters specify the *TokenCenter* service location and the number of tokens to be returned to the *TokenCenter*.

An example of using a *BrokerCenter* is the implementation of the *Interrupt Handler Task* and *Task 1* where *BrokerCenter* is used for publish/subscribe communication. In the presented *PIEthon* code, the sensor reading acquired from *Sensor0* is used to trigger the execution of the rest of the application. The *Interrupt Handler Task* is publishing information to a *BrokerCenter* using a *BrokerCentLib.Publish* method. The parameters specify the *BrokerCenter* service location and the announcement that will be published. *Task 1* subscribes to the *BrokerCenter* using a *BrokerCentLib.Subscribe* method. The parameters specify the *BrokerCenter* service location, the terms of the subscription, and the location of the *Broker* service. To keep the *BrokerCenter* application-independent service, we use three-party *publish/subscribe/interpret* communication model, instead of traditional two-party *publish/subscribe* model. Third parties included into the model are broker services used for interpretation of the published announcements and matching them with subscriptions [16, 18]. In given example, *Controller0* is used as a broker service.

In similar way as we extended Python to *PIEthon*, any system or scripting language can be extended into an event-driven service composition language. To transform an ordinary programming language into its SOA&EDA-ready counterpart, one needs to develop a collaboration library for target language that supports invocation of *Web Services* and *coopetition services*.

## 7 HUSKY: TABULAR SOA&EDA LANGUAGE

Although widely used as Python, *PIEthon* is still intended for users with advanced programming skills. However, as CPS become more ubiquitous, they often need to be personalized towards specific user expectations. To allow end-users and automation practitioners who may not be educated in software engineering the development of ubiquitous CPS applications, we have designed a programming language that targets these groups of users. During the design of a new language, we had two main objectives. First, a new language should exhibit a computational model suitable for non-programmers. Second, it should provide the design workspace that facilitates the development of CPS applications that include multiple event flows between multiple automation tasks and enable a comprehensive representation and management of automation logic.

*PIEthon* programming is based on imperative computation model. Programs are based on series of statements that execute sequentially in time. Program state is kept in variables which are explicitly defined and managed by language statements. Furthermore, each task is implemented in a separate program workspace, which does not allow a comprehensive view of the entire automation process. As shown in Fig. 3, composing services into environment automation application involves dealing with multiple event flows in distributed environment. Comparing presentations of the CPS control process implemented in *PIEthon* with the application outline shown in Fig. 3, it is obvious that *PIEthon* is falling short of comprehensive view of task interrelations that are clearly visible in Fig. 3. This makes the process of building and maintaining the CPS, which requires constant changes in application logic, hard and error-prone.

Describing and controlling complex event-driven systems from end-user applications requires a paradigm shift that extends the basic principles of imperative model. To recognize key factors that bring the programming close to non-programmers, we studied the most successful computing paradigms and languages implemented in information systems. We found that spreadsheets are the most widely used languages in today's end-user applications. They are understood and used by millions of users every day for building custom calculations and data representations. Furthermore, two-dimensional organization of spreadsheet programs provides a design workspace that visually reflects the logical structure of the CPS. Based on these assumptions, we have designed a new tabular language for service composition on top of *PIEthon*. The new language is called *HUSKY*: *HUman-centered Service composition worKspace and methodologY*. Just as husky dogs are harnessed into a team to drag a sled, so *HUSKY* connects devices virtualized as services into CPS application.

### 7.1 Tabular HUSKY Workspace

*HUSKY* combines the features of textual and visual languages in a form of tabular representation. We reused basic spreadsheet concepts, such as cells, cell values, and cell references, and extended them with declarations of *environmental* and *coopetition services*, as well as imperative constructs for their invocations. Figure 7 shows a layout of the CPS application shown in Fig. 3 in *HUSKY* table. Cells contain task definitions, coopetition service instances, environmental service definitions, and constant values. Tasks are defined by set of events. Each event is defined in separate table cell. Events invoke environmental services and enable collaboration between tasks. Cells [C5-C10], [G1-G6], and [G8-G13] contain the events comprising *Task 1-3*, respectively. *Interrupt Handler Task* is implemented in cells [C1-C3]. Cell [E4] contains an instance of the *TokenCenter* coopetition service, while cell [E8] contains an instance of the *Queue* coopetition service. An instance of the *BrokerCenter* is given in cell [A2], while broker service is defined in cell [A5]. The rest of the cells contain either environmental service definitions or constant

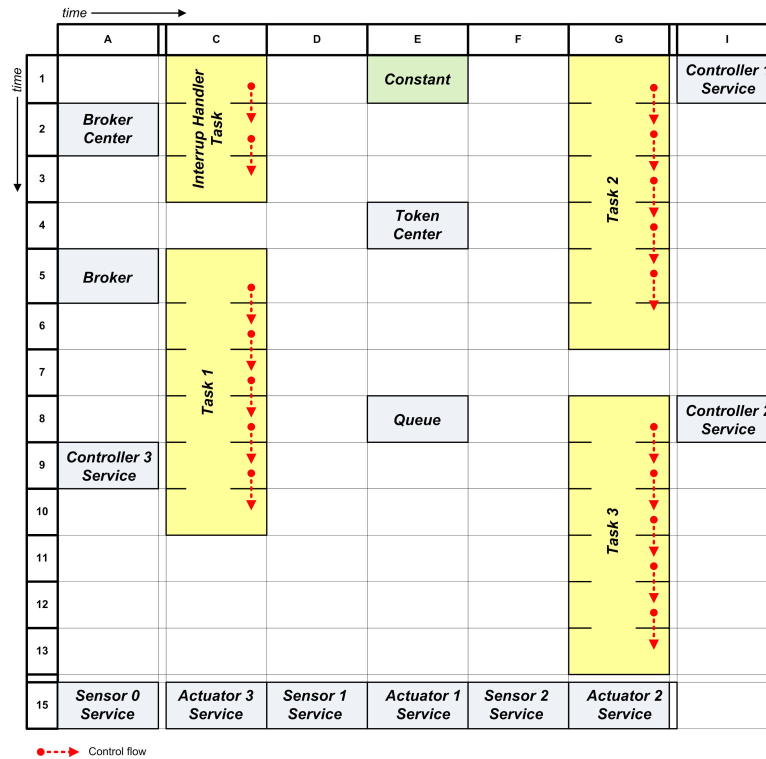


*Fig. 7. A two-dimensional tabular HUSKY workspace*

values (cell [E1]). Environmental service definitions are used to identify the sensor, actuator, and controller devices.

*HUSKY* environment provides an intuitive paradigm for expressing concurrency in service composition. We introduce the time ordering relation, which transforms the spatial organization of events in cells into their ordering in time. The time ordering relation within a *HUSKY* workspace is defined along two dimensions: horizontal and vertical. Time progresses from left to right and from top to bottom simultaneously. Two events are sequential in time if they occupy two adjacent cells in a *HUSKY* workspace. A set of adjacent cells makes a sequence of events, while empty cells partition the workspace into temporally independent event sequences. This enables a spatial organization of four independent tasks in the same way as outlined in Fig. 3. Each task is a sequence of events temporally independent from any other tasks. Control flow arrows presented in Fig. 8 denote time ordering relation during the execution of events that comprise given task. Separation of event sequences with empty cells enables modeling of multiple event flows between a set of event-driven tasks in a single workspace.

The *HUSKY* language is based on simple graphical presentation of process logic, which presents CPS control tasks and their interrelationships in a unified workspace, similar to the scheme given in Fig. 3. The unified workspace is defined in 2D table and presents the overall CPS control process and significantly simplifies the design of CPS application. The representation of service composition is organized in a two-dimensional space [25] that resembles the process of sketching an idea on a sheet of paper. We have found the graphical representation in tabular space as the most suitable form of program representation for developers that may not be experienced in event-driven programming, because even expert developers find it easier and less time consuming to organize event-driven programs in graphical than textual form. Tabular workspace enables visual organization of tasks in a way similar to graphical schemes presented in Fig. 3, yet easily readable on the screen. Second, the two-dimensional space based on cells, cell references, and cell values as basic data handling elements do not require the use of variables [26]. In *HUSKY*, users use cells as containers for values and definitions, and reference them with the cell indices as in any other spreadsheet-based language [27].

### 7.2 Application Implementation in HUSKY

Figure 8 presents the complete *HUSKY* implementation of the CPS application shown in Fig. 3 and outlined in Fig.

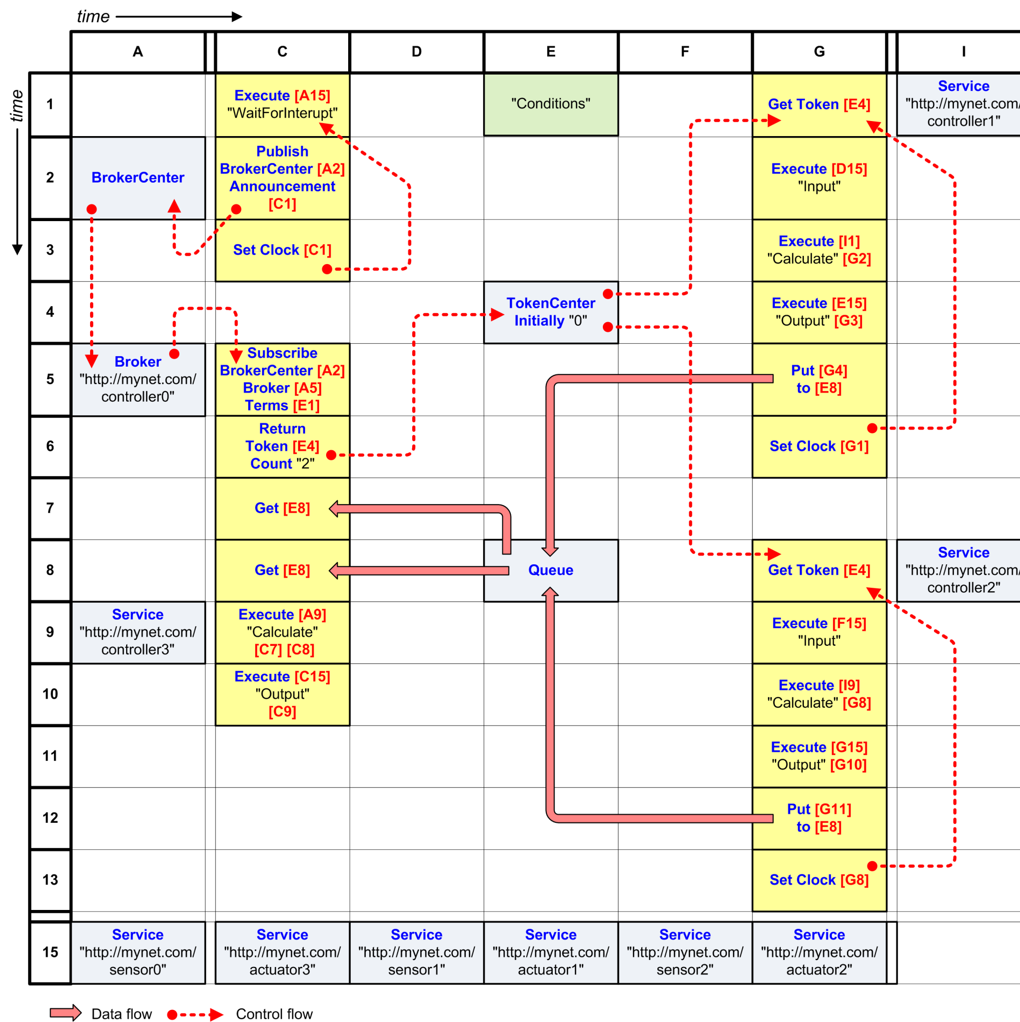


*Fig. 8. HUSKY implementation of CPS application*

7.

The application implements the following event flow. The *Interrupt Handler Task* defined in cells [C1-C3] invokes *Sensor0* and waits for a sensor reading. When a reading is acquired, it is published as an announcement to the *BrokerCenter* service defined in cell [A2]. If the published announcement satisfies the subscription terms defined in cell [E1], the execution of the *Task 1* defined in cells [C5-C10] is triggered. When started, *Task 1* uses the *TokenCenter* defined in cell [A2] to activate the execution of the *Task 2* in cells [G1-G6] and *Task 3* in cells [G8-G13]. *Tasks 2* and *3* invoke their sensor devices *Sensor1* and *Sensor2* defined in cells [D15] and [F15] and acquire a sensor reading, process the reading by invoking *Controller1* and *Controller2* services defined in cells [I1] and [I8], and output the result by invoking *Actuator1* and *Actuator2* services defined in cells [E15] and [G15]. They use the instance of *Queue* service defined in cell [E8] to send the sensor readings to *Task 1*. After retrieving the sensor readings, the *Task1* proceeds to process the readings using the *Controller3* service defined in cell [A9] and output the result using the *Actuator3* service defined in cell [C15].

### 7.3 HUSKY Language Constructs

The following sections describe the most important *HUSKY* language constructs specified within the *HUSKY* cells. These include the definition of environmental services, instantiation of coopetition services, and definition of events for service invocation, publish-subscribe messaging, communication, synchronization, and several additional constructs.

#### 7.3.1 Service invocation

Service invocation events are denoted by the *Execute* keyword. Cell [G3] in Fig. 8 shows an example of service invocation in *HUSKY*. To invoke a service, the service location and the operation name must be specified. For example, cell [G3] contains the event for invocation of the *Controller1* service:

**Execute** [I1] "Calculate" [G2]

where *Execute* determines that the cell contains a service invocation event, [I1] specifies a cell in which the service location is defined, "Calculate" specifies the operation that the invoked service should perform, and [G2] specifies the cell that contains service invocation parameters. After execution of the *Execute* event in cell [G3], this cell will contain the result of execution of the *Controller1* service. The result of the *Execute* event in cell [G3] is used as the input parameter for the execution of the *Execute* event in cell [G4].

To keep the syntax of the service invocation event compact, the service location is specified in a separate cell. For example, cell [I1] contains the definition of the *Controller1* service:

**Service** "http://mynet.com/controller1"

where *Service* determines that the cell contains a service definition, while "*http://mynet.com/controller1*" is the endpoint URL where the service representing the device is available.

#### 7.3.2 Communication

To enable asynchronous communication between tasks, the *Queue* service is used. The instance of *Queue* service is defined in the cell [E8], using the **Queue** keyword. Cell [G5] contains an event for sending a message to the *Queue*:

**Put** [G4] **to** [E8]

where *Put* and *to* are keywords for this type of event. The parameter [G4] specifies the cell that contains the message, while the parameter [E8] specifies the cell where the *Queue* is instantiated.

Cells [C7] and [C8] contain events for reading a message from the *Queue*:

**Get** [E8]

where *Get* is a keyword for the event for acquiring a message from a *Queue*. The parameter [E8] specifies the cell in which the *Queue* is instantiated. After execution of these two events, cells [C7] and [C8] will contain the result of the *Get* event.

#### 7.3.3 Synchronization

To synchronize the execution of concurrent automation tasks, the *TokenCenter* service is used. We define an istance of *TokenCenter* service in cell [E3]:

**TokenCenter Initially** "0"

where the *TokenCenter* keyword represents the instance of a *TokenCenter* service, while *Initially* defines the initial number of tokens.

Synchronization consists of two events: *Get Token* and *Return Token*, which acquire tokens from and return them back to the *TokenCenter*.

Cells [G1] and [G8] contain a *Get Token* event:

**Get Token** [E3]

where *Get Token* determines an event for acquiring a token from the *TokenCenter*. The parameter [E3] specifies the cell in which the *TokenCenter* is instantiated. If a token is not available in *TokenCenter*, execution of the sequence will be suspended until one becomes available through execution of the *Return Token* event elsewhere in the *HUSKY* workspace.

Cell [C6] contains a *Return Token* event:

**Return Token** [E4] **Count** "2"

where *Return Token* determines an event for returning tokens to the *TokenCenter*. The parameter [E4] specifies the cell in which the *TokenCenter* is defined, while *Count* specifies the number of tokens to be returned to the *TokenCenter*.

#### 7.3.4 Publish-Subscribe Messaging

For content-based messaging using publish/subscribe paradigm, the *BrokerCenter* service is used. The instance of the *BrokerCenter* service is defined in the cell [A2]. An example of a *Broker* service definition is given in cell [A5]:

**Broker** "http://mynet/controller0"

The *Broker* keyword defines a *Broker* service, while the literal "*http://mynet/controller0*" represents the URL where the *Broker* service is available.

The cell [C5] defines an example of the *Subscribe* event:

**Subscribe BrokerCenter** [A2]

**Broker** [A5] **Terms** [E1]

The parameter [A2] specifies the cell that contains the instance of the *BrokerCenter* service, the parameter [A5] specifies the cell that contains the definition of the *Broker* service, while the parameter [E1] specifies the cell that contains the terms of the subscription.

The cell [C2] defines an example of *Publish* event:

**Publish BrokerCenter** [A2] **Announcement** [C1]

The parameter [A2] specifies the cell that contains the instance of the *BrokerCenter*, while the parameter [C1] specifies the cell that contains the announcement that will be published.

#### 7.3.5 Event Execution Control Flow

Since event sequences are temporally independent, the execution of each sequence proceeds within its local time which is controlled by a local clock. Ticks of the clock are aligned with the cells that comprise the sequence. The local clock makes a tick each time an event (i.e. cell) of the sequence is executed. For example, the local clock of sequence [C1-C3] in Fig. 8 makes the following ticks: C1, C2, and C3. Cell [C3] contains special event *Set Clock* which is used to control the execution flow of the event sequence:

**Set Clock** [C1]

where *Set Clock* determines an event for controlling the ticks of the sequence's local clock. The parameter [C1] specifies the cell to which the flow of control will be redirected after the execution of the event.

In addition to enforcing unconditional control flow, the local sequence clock can be controlled using a conditional control flow

**If** [C4] **Set Clock** [C1]

where [C4] specifies a cell that contains a service invocation event that evaluates to Boolean value, while [C1] specifies the cell to which the flow of control will be redirected if the referenced cell contains the Boolean value *true*.

### 7.4 HUSKY to PIEthon Translation Framework

Figure 9 presents the *HUSKY* translation and execution framework that consists of the *HUSKY Editor* and the *Execution Environment*. The *HUSKY Editor* is a user-friendly GUI for development of event-driven CPS applications based on service composition. The *Execution Environment* consists of host machines and devices that participate in the execution of CPS applications written in *HUSKY Editor*.

Figure 9 shows the execution environment which consists of a set of interpreters [28-30], coopetition services, and environmental services. *HUSKY Compiler* translates the service composition to a set of *PIEthon* tasks. For example, we translate the application presented in Fig. 8 to a set of *PIEthon* tasks shown in Fig. 6.

Coopetition services are stored in the *Coopetition Service Repository* [31, 32] and dispatched on network machines before the execution of *PIEthon* tasks. Before tasks are scheduled on the interpreters, *Coopetition Extractor* analyzes the coopetition service invocations and generates the *coopetition service list*. This list includes references to all coopetition services used in application and references to hosts where they must be executed. Using the coopetition service list, the *Dispatcher* obtains executable *coopetition services code* from the repository, and deploys and executes it on target hosts.

Once the coopetition services are deployed, *PIEthon* tasks are scheduled and executed on the machines of the computer network that host Python interpreters. The *Coopetition libraries* run on top of interpreters and provide the support for invocation of coopetition and environmental services from Python. Finally, devices of the sensor, actuator, and controller network are virtualized and made accessible through a set of *environmental services* that execute on host machines.

### 7.5 Cognitive Evaluation of HUSKY from End-User Perspective

This section presents analysis and comparison of cognitive features of textual, graphical, and *HUSKY* programming languages for service composition. The focus of the analysis is the simplicity and cognitive burden imposed on users of each group of languages. Languages have been evaluated using *Cognitive Dimensions framework* [33]. This framework enables designers of information system to analyze and evaluate system's features from the end-user's perspective. To date, the framework has been successfully applied for evaluation of visual interfaces and programming languages. The framework is not tied to any particular application domain, but is based on psychological factors that impact human mental performance during interaction with any kind of notational system. Therefore, it enables qualitative comparison of programming languages with different properties, intended application domain, and targeted user base.

Using *Cognitive Dimensions framework*, we analyze syntactic and semantic properties of developed

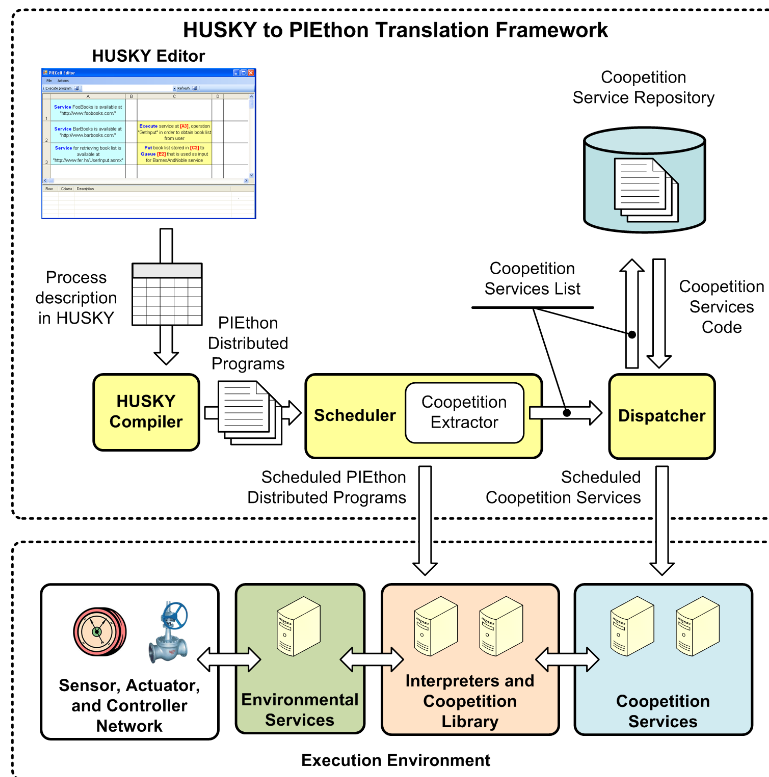


*Fig. 9. HUSKY translation and execution*

programming languages. Cognitive analysis of syntactic properties deals with the number of elementary language elements, their representation on the screen, the technique to compose a set of elementary elements into an application, and the ease of manipulation with elementary elements in order to create or modify an application. Cognitive analysis of semantic properties deals with closeness of visual representation of language elements to the real-world concepts they represent and inspects how a constellation of elementary elements in a workspace define the program behavior in terms of data flow, control flow, and concurrency. Cognitive analysis of languages presented in this paper has been made using following set of cognitive dimensions: *consistency*, *closeness of mapping*, *abstraction*, *visibility*, *viscosity*, and *hidden dependencies*. The results are presented in Table 1.

*Visual notation*. Visual notation of programming languages is evaluated using closeness of mapping, abstraction, visibility, and viscosity. Textual languages, like scripting languages, are based on simple lexical and syntactic features resembling written form of natural language. As a result, these languages have fair closeness of mapping and abstraction level. On the other hand, XML-based languages, although still based on textual notation, use special machine-readable markup and syntactic structure that have negative impact on closeness of mapping and abstraction level. As the complexity of service composition grows, textual representation becomes less visible and more difficult to change. Graphical languages use glyphs and icons that map closely to users' cognitive models and have good level of abstraction. However, the graphical descriptions become difficult to manage as the complexity scales because of an increasing number of intersecting arcs, which results in a cluttered view. *HUSKY* combines the best notational elements from both textual and graphical languages. Textual elements are used to convey semantics of language statements which results in good closeness of mapping and abstraction. Furthermore, just as in spreadsheet languages, tabular form provides simple layout with good visibility and fair viscosity features.

*Temporal properties*. Temporal properties of programming languages are evaluated using closeness of mapping, abstraction, visibility, and hidden dependencies. Textual languages are based on one-dimensional flow of time which is aligned with users' cognitive models of time. However, the use of multitasking models that incurs multiple interwoven control and event flows

*Table 1. HUSKY Cognitive Dimensions Evaluation*

| | | TXT | GRAPH | HUSKY |
|---|---|---|---|---|
| Visual notation | CM | ◑ | ● | ● |
| | A | ◑ | ● | ● |
| | VIS | ○ | ◑ | ● |
| | VSC | ○ | ◑ | ◑ |
| Temporal properties | CM | ● | ◑ | ● |
| | A | ● | ● | ● |
| | VIS | ◑ | ◑ | ● |
| | HD | ◑ | ○ | ◑ |
| Control flow | C | ○ | ◑ | ● |
| | A | ○ | ● | ● |
| | VIS | ◑ | ◑ | ● |
| | HD | ◑ | ◑ | ◑ |
| Data flow | CM | ○ | ● | ● |
| | A | ○ | ● | ● |
| | VIS | ◑ | ◑ | ◑ |
| | HD | ○ | ◑ | ◑ |
| Comments | CM | ● | ● | ● |
| | A | ● | ◑ | ● |
| | VIS | ● | ◑ | ● |

Cognitive dimension: Consistency (C), Closeness of mapping (CM), Abstraction (A), Visibility (VIS), Hidden dependencies (HD), Viscosity (VSC)

Level of support: ○ - Poor ◑ - Fair ● - Good

in one dimension lowers visibility and incurs hidden dependencies. Graphical languages use a model based on free two-dimensional flow of time. This model is based on using arcs connecting tasks and defining temporal relationships which can be well understood by users. However, larger graphical descriptions often suffer from low readability because of free-form layout and crossed or overlapping arcs, and thus incur low visibility and hidden dependencies. *HUSKY* uses temporal model which is two dimensional but does not use free-form layout as in graphical languages. By restricting flow of time to horizontal and vertical directions, *HUSKY*'s flow of time is suitable to be modeled in tabular space which is simple to understand and manage. Moreover, tabular form enables simple and visible presentation of temporal dependencies with fair level of hidden dependencies.

*Control flow*. Control flow of programming languages is evaluated using consistency, abstraction, visibility, and hidden dependencies. Textual languages use specialized statements for managing the control flow. The use of these statements breaks the basic principles of unidirectional, continuous, and one-dimensional flow of time. The statements can divert the flow back in time or into the future and cause gaps in the control flow. Thus, the use of these statements is inconsistent with basic model of time. Large compositions may have many control flow statements that divert the control flow in complex ways and cause lower level of visibility and hidden dependencies. Control flow in graphical languages is described by defining temporal precedence of statements with connecting arcs. Arcs are consistent with two-dimensional flow of time and provide a good basic construct for managing control flow. However, basic arcs cannot describe complex patterns like conditional branching and often additional symbols are used. Furthermore, control flow description quickly loses its visibility and includes hidden dependencies for complex descriptions. Control flow in *HUSKY* is based on statements that extend basic concepts behind tabular form, cells, and cell references in a consistent way. The control flow paradigm is based on clock ticks abstraction which is aligned with real-world clocks. Furthermore, control flow statements include cell references which are visible as cell contents. This has positive effect on overall visibility and lowers hidden dependencies.

*Data flow*. Data flow of programming languages is evaluated using closeness of mapping, abstraction, visibility, and hidden dependencies. Variables are the basic constructs for building dataflow in textual languages. Variables are abstract mathematical constructs that are not aligned with end-users' mental models. Furthermore, overall dataflow is implicitly described as series of data transformation mediated through variables. This reduces visibility and imposes hidden dependencies. In graphical languages, data-flow dependencies are expressed using specialized arcs that are consistent with the basis of the notation. Describing data-flow using arcs is a simple paradigm understood by end-users. However, complex descriptions often exhibit lower visibility due to many overlapping arcs and include hidden dependencies. *HUSKY* data flow is directly supported by extension of tabular form where variables are replaced by cells, and data flow is expressed by referencing cells as in any spreadsheet language. *HUSKY* is based on simple abstractions understood by many spreadsheet users. Moreover, tabular form enables fair visibility and reduces hidden dependencies.

*Comments*. Comments in programming languages are evaluated using closeness of mapping, abstraction, and visibility. Textual languages enable comments written in natural language. The use of natural language exhibits good closeness of mapping, simple abstractions, and visibility. Graphical languages enable use of auxiliary symbols and text for describing user comments. Auxiliary symbols impose new abstractions and lower the overall visibility of the description. *HUSKY* enables users to mix-in the comments between statements placed in each

cell. Thus, users are able to use comments to increase visibility of the description. For example, the statement *Put* [G4] *to* [E8] placed in cell [G5] in Fig. 8 augmented with inline comments may look like this:

**Put** *temperature* [G4] *from living room thermometer*
**to** *air condition feedback queue* [E8]

The *HUSKY*-style inline comments unobtrusively complement the keywords that make service compositions more readable.

## 8 CONCLUSION

In this paper, we present *service composition* as a paradigm for development of cyber-physical systems (CPS) that integrate physical world of devices with virtual world of software-controlled automation processes. We propose an *event-driven service-oriented architecture*, where devices, such as sensors, controllers, and actuators, appear as *environmental services* that are linked into automation applications using *event-driven service composition languages*. To enable development of event-driven automation processes, special-purpose event-handling *coopetition services* are developed as fundamental components of the architecture.

We have designed a suite of *event-driven service composition languages* that enable development of CPS control processes on different levels of abstraction. To build a language suitable for wide programmer community, we propose a transformation of a general-purpose scripting language into an event-driven service composition language. As an example, we extended Python with modules for invocation of Web Services and handling events through coopetition services.

While upgraded Python is convenient for use by a wide community of professional programmers, end-users and automation practitioners still find this language complex and intractable. The main obstacle of textual languages is their hard perception from the point of automation process design. Event-driven automation processes are often sketched on a paper using graphical schemes. To augment an automation language with a graphical design workspace, we design semi-graphical end-user language *HUSKY* (*HUman-centered Service composition worKspace and methodologY*) that resemble design methodology usual in automation practice. *HUSKY* enables a comprehensive user-friendly representation and management of automation logic in two-dimensional tabular workspace. *HUSKY* programs resemble the automation process sketched as a graphical scheme on a sheet of paper. Cognitive evaluation of *HUSKY* against textual and graph-based service composition languages shows its advantages with respect to consistency, closeness of mapping, level of abstraction, hidden dependencies, visibility, and viscosity.

To enable a seamless cooperation between application developers with different skills and knowledge, we define a multistage process for translation of automation logic from high to low level of abstraction. The high level application specification written in *HUSKY* is translated into *PIEthon*, which is then executed on Python interpreter augmented with modules for invocation of Web services and handling of events through coopetition services. Multistage translation enables cooperation between end-users and professional programmers. For example, end-users may define the core automation logic in *HUSKY*. After being translated into the low-level and more expressive language, such as *PIEthon*, professional programmers may augment the core automation process with additional logic, such as conversion of service parameter data formats, which is required for correct execution of service compositions.

Future work in the field requires further opening of CPS design towards end-users. New, consumer-oriented CPS design paradigms are needed to manage diversity and amount of environmental stimuli on one side, and meet individual user preferences caused by vast number of imaginable use cases on the other. In [34], authors propose an application development paradigm that extends GUI from a well-known "language of interaction" into an application development language. Because of its suitability for users not trained in software engineering, this paradigm may serve as a starting point towards future consumer-driven CPS development.

## ACKNOWLEDGMENT

The authors acknowledge the support of the Ministry of Science, Education, and Sports of the Republic of Croatia through research project Computing Environments for Ubiquitous Distributed Systems (036-0362980-1921). Furthermore, the authors wish to thank Ivan Zuzak, Klemo Vladimir, Marin Silic, Goran Delac, Jakov Krolo, Ivan Budiselic, and Josko Radej from School of Electrical Engineering and Computing at University of Zagreb, Daniel Skrobo, Ivan Skuliber, and Ivan Benc from Ericsson Nikola Tesla Zagreb, Ivan Gavran from Calyx Zagreb, Franjo Plavec and Ivan Matosevic from University of Toronto, Ivo Krka from University of Southern California Los Angeles, and Boris Debic from Google, Inc. for their help with preparing this paper. Finally, many thanks to student members of our research project who participated in the implementation of the *HUSKY* framework.

## REFERENCES

[1] E. A. Lee, “Cyber physical systems: Design challenges,” in *Proceedings of the International Symposium on*

*Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008.

[2] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-based Internet of Things: Discovery, query, selection, and on-demand provisioning of web services," *IEEE Transactions on Services Computing*, vol. 3, pp. 223–235, July 2010.

[3] A. S. Taylor, R. Harper, L. Swan, S. Izadi, A. Sellen, and M. Perry, "Homes that make us smart," *Personal Ubiquitous Computing*, vol. 11, pp. 383–393, June 2007.

[4] S. H. Park, S. H. Won, J. B. Lee, and S. W. Kim, "Smart home – digitally engineered domestic life," *Personal Ubiquitous Computing*, vol. 7, pp. 189–196, July 2003.

[5] Z. Laliwala and S. Chaudhary, "Event-driven service-oriented architecture," in *Proceedings of the International Conference on Service Systems and Service Management*, (Melbourne, Australia), pp. 1–6, July 2008.

[6] "Event-driven SOA: A better way to SOA," tech. rep., TIBCO Software Inc., 2006.

[7] J.-L. Marechaux, "Combining service-oriented architecture and event-driven architecture using an enterprise service bus," tech. rep., IBM DeveloperWorks, 2006.

[8] S. Bharti *et al.*, "Fine grained SEDA architecture for service oriented network management systems," *International Journal of Web Services Practices*, vol. 1, no. 1-2, pp. 158–166, 2005.

[9] "Service component architecture – unifying SOA and EDA," tech. rep., Fiorano Software Technologies, 2010.

[10] J. Hanson, "Event-driven services in SOA: Design an event-driven and service-oriented platform with Mule," tech. rep., JavaWorld.com, 2005.

[11] N. Milanovic, "Service engineering design patterns," in *Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering*, SOSE '06, (Washington, DC, USA), pp. 19–26, IEEE Computer Society, 2006.

[12] "Transducer electronic data sheets (TEDS)," tech. rep., NIST.

[13] K. Czajkowski *et al.*, "The WS-Resource Framework," tech. rep., Globus Alliance, 2004.

[14] "Service-oriented ecosystem," tech. rep., EUREKA ITEA 05022 SODA, 2010.

[15] "Devices profile for web services (DPWS)," tech. rep., OASIS Standard, 2009.

[16] A. Milanovic, S. Srbljic, D. Skrobo, D. Capalija, and S. Reskovic, "Coopetition mechanisms for service-oriented distributed systems," in *Proceedings of 3rd International Conference on Computing, Communications and Control Technologies Cybernetics and Informatics (CCCT'05)*, (Austin, Texas, USA), pp. 118–123, July 2005.

[17] S. Reskovic, "Synchronization and communication mechanisms for service-oriented applications," b.sc. thesis, University of Zagreb, School of Electrical Engineering and Computing, 2005.

[18] D. Capalija, "Publish/subscribe mechanisms for implementation of content-based networking," b.sc. thesis, University of Zagreb, School of Electrical Engineering and Computing, 2005.

[19] S. Srbljic, D. Skvorc, and D. Skrobo, "Programming language design for event-driven service composition," *Automatika*, vol. 51, pp. 374–386, Dec. 2010.

[20] I. Gavran, A. Milanovic, and S. Srbljic, "End-user programming language for service-oriented integration," in *Proceedings of 7th Workshop on Distributed Data and Structures*, (Santa Clara, CA, USA), pp. 118–123, Jan. 2006.

[21] I. Gavran, "End-user language for service-oriented programming model," m.sc. thesis, University of Zagreb, School of Electrical Engineering and Computing, 2006.

[22] A. Alves *et al.*, "Web services business process execution language (WS-BPEL) 2.0," tech. rep., OASIS, 2006.

[23] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services decription language (WSDL) 1.1," tech. rep., World Wide Web Consortium (W3C), 2001.

[24] M. Ben-Ari, *Principles of concurrent and distributed programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[25] D. J. Power, "A history of microcomputer spreadsheets," *Communications of the Association for Information Systems*, vol. 4, pp. 154–162, Oct. 2000.

[26] J. Sajaniemi, "Visualizing roles of variables to novice programmers," in *Proceedings of the 14th Annual Workshop of the PPIG'02*, (London, UK), pp. 111–127, June 2002.

[27] R. Pardo *et al.*, "Process and apparatus for converting a source program into an object program," tech. rep., U.S. Patent 4,398,249, Aug. 1983.

[28] D. Skrobo, A. Milanovic, and S. Srbljic, "Distributed program interpretation in service-oriented distributed systems," in *Proceedings of the WMSCI'05*, (Orlando, FL, USA), pp. 193–197, July 2005.

[29] A. Milanovic, *Service-Oriented Programming Model*. Ph.d. thesis, University of Zagreb, School of Electrical Engineering and Computing, 2005.

[30] D. Skrobo, "Distributed program interpretation in service oriented architectures," m.sc. thesis, University of Zagreb, School of Electrical Engineering and Computing, 2006.

[31] M. Podravec, I. Skuliber, and S. Srbljic, "Service discovery and deployment in service-oriented computing environment," in *Proceedings of the WMSCI'05*, vol. 3, (Orlando, FL, USA), pp. 5–10, July 2005.

[32] M. Podravec, "Service discovery and deployment in service-oriented systems," m.sc. thesis, University of Zagreb, School of Electrical Engineering and Computing, 2006.

[33] T. R. G. Green, A. E. Blandford, L. Curch, C. R. Roast, and S. Clarke, “Measuring cognitive load – a solution to ease learning of programming,” in *Proceedings of the World Academy of Science, Engineering, and Technology*, vol. 20, pp. 216–219, Apr. 2007.

[34] S. Srbljic, D. Skvorc, and D. Skrobo, “Widget-oriented consumer programming,” *Automatika*, vol. 50, pp. 252–264, Dec. 2010.

**Siniša Srbljić** is a Professor at the University of Zagreb, School of Electrical Engineering and Computing. His research career also spans Canada and USA where he was visiting the University of Toronto, the AT&T Labs, San Jose, the University of California, Irvine, and Futurewei Technologies, Inc., Center for Innovations, Corporate Research US Huawei, Santa Clara. As part of a Google Research Award grant, he led a research team that designed a consumer programming tool Geppeto, which enables consumers without formal education in computer engineering to develop their own software applications. He is head of Consumer Computing Laboratory at University of Zagreb, School of Electrical Engineering and Computing. His research interests include: consumer computing; collaborative intelligence; programming language translation; theory of computing.

**Dejan Škvorc** is a research and teaching assistant, and member of the Consumer Computing Laboratory at School of Electrical Engineering and Computing, University of Zagreb, Croatia. He received his B.Sc. degree in 2003, M.Sc. degree in 2006, and PhD in 2010 from School of Electrical Engineering and Computing, University of Zagreb. During 2007, Dejan Skvorc spent four months as a software engineering intern in Google's Mountain View office, CA, USA, with Google Gadgets group. He is a coauthor and one of the architects of the Google's inter-gadget communication framework. His research interests include service-oriented architectures, programming language design, end-user development, and consumer programming.

**Miroslav Popović** is a computer science Ph.D. candidate and research assistant at the School of Electrical Engineering and Computing, University of Zagreb. He received his M.Sc. degree in 2006 from School of Electrical Engineering and Computing, University of Zagreb. He was a software engineering intern at Google's Mountain View office in CA, USA. His interests include service-oriented architectures and end-user language development. Currently, he is working on design of consumer program synchronization mechanisms.

## AUTHORS' ADDRESSES

**Prof. Siniša Srbljić, Ph.D.**
**Dejan Škvorc, Ph.D.**
**Miroslav Popović**
**Department of Electronics, Microelectronics, Computer and Intelligent Systems,**
**Faculty of Electrical Engineering and Computing,**
**University of Zagreb,**
**Unska 3, HR-10000, Zagreb, Croatia**
**email: sinisa.srbljic@fer.hr, dejan.skvorc@fer.hr, miroslav.popović@fer.hr**

Received: 2011-06-03
Accepted: 2012-02-20