ISSN 0005-1144
ATKAFF 51(4), 374–386(2010)

Siniša Srbljić, Dejan Škvorc, Daniel Skrobo

Programming Language Design for Event-Driven Service Composition

UDK 004.434:65
IFAC 2.8.1

Original scientific paper

To adapt to rapidly changing market conditions and increase the return of investment, today's IT solutions usually combine service-oriented architecture (SOA) and event-driven architecture (EDA) that support reusability, flexibility, and responsiveness of business processes. Programming languages for development of event-driven service compositions face several main challenges. First, a language should be based on standard service composition languages to be compatible with SOA-enabling technologies. Second, a language should enable seamless integration of services into event-driven workflows. Third, to overcome a knowledge divide, language should enable seamless cooperation between application developers with different skills and knowledge.

Since WS-BPEL is widely accepted as standard executable language in SOA, we extended WS-BPEL with support for event-driven workflow coordination. We designed event-handling mechanisms as special-purpose *Coopetition services* and augmented WS-BPEL with primitives for their invocation. *Coopetition services* augment SOA with fundamental EDA characteristics: decoupled interactions, many-to-many communication, publish/subscribe messaging, event triggering, and asynchronous operations. To make the application development familiar to wide community of developers, we designed an application-level end-user language on top of WS-BPEL whose primitives for invocation of regular Web services and *Coopetition services* resemble the constructs of typical scripting and coordination language.

Key words: Service composition, Service-oriented event-driven programming, Programming language design

Oblikovanje programskih jezika za događajima poticanu kompoziciju usluga. S ciljem prilagodbe promjenjivim tržišnim uvjetima i povećanja isplativosti ulaganja, današnji informacijski sustavi grade se spregom uslužno usmjerene i događajima poticane arhitekture koje omogućuju oblikovanje višestruko iskoristivih i prilagodljivih poslovnih procesa s mogućnošću odziva na pojavu događaja. Programski jezici za događajima poticanu kompoziciju usluga pokazuju nekoliko glavnih značajki. Prvo, jezik mora naslijediti svojstva standardnih jezika za kompoziciju usluga kako bi bio sukladan tehnologijama uslužno-usmjerene arhitekture. Drugo, jezik mora omogućiti prirodni način povezivanja usluga u događajima poticane poslovne procese. Treće, razvijateljima različitih znanja i vještina potrebno je osigurati mogućnost udruženog sudjelovanja u razvoju primjenskih programa.

Budući da je WS-BPEL standardni jezik za kompoziciju usluga, izabran je kao osnovica za oblikovanje jezika za događajima poticanu kompoziciju usluga. Oblikovan je poseban skup *usluga suradnje i natjecanja* kojima je uslužno-usmjerena arhitektura proširena elementima događajima poticane arhitekture, kao što su međudjelovanje zasnovano na slaboj povezivosti, komunikacija u grupi, objava/pretplata, reakcija na pojavu događaja i asinkrone operacije. Jezik WS-BPEL proširen je programskim primitivama za pozivanje tih usluga. Kako bi se razvoj primjenskih programa približio širokoj zajednici graditelja programske potpore, povrh jezika WS-BPEL oblikovan je primjenski jezik za krajnjeg korisnika čije primitive za pozivanje primjenskih usluga te *usluga suradnje i natjecanja* nalikuju naredbama skriptnih i koordinacijskih jezika.

Ključne riječi: kompozicija usluga, programiranje zasnovano na događajima poticanoj kompoziciji usluga, oblikovanje programskih jezika

1 INTRODUCTION

During the last couple of years, there were a lot of discussions related to how service-oriented architecture (SOA) and event-driven architecture (EDA) fit together. While some of disputants say that SOA and EDA go together nicely, others claim that they are competing software architectures [1]. In latest discussions, however, software architects agreed that SOA and EDA are two complementary software design paradigms, which, when combined together, may successfully address complex integration challenges [2, 3].

Today's business applications are rarely deployed and used in isolation. Instead, they are connected with other applications in order to create integrated business solutions that span across organizational boundaries. To be able to adapt to rapidly changing market conditions and increase the return of investment, organizations require IT infrastructure that supports reusability, flexibility, and responsiveness. Therefore, organizations are seeking for architectures and technologies that provide the ability to break monolithic applications into reusable software components, and to compose these components into flexible business workflows that proactively respond to events from inside and outside of the application.

Service-oriented architecture (SOA) is an architectural concept that enables reuse of existing software functionalities in various types of business applications and integration of heterogeneous software components into coherent business solutions. In SOA, software functionalities are exposed to consumers and other applications as services accessible over a network through standardized interfaces. Through standardization, SOA enables seamless integration of heterogeneous software components, regardless of the hardware platform, the operating system, and the programming language used for their implementation. In an environment where software components are exposed as services, *service composition* is used as a design paradigm to connect mutually independent services into business-specific workflows [4]. *Service composition languages* are, therefore, used as process description languages to define business processes from which the execution of services is orchestrated. To be SOA-ready, service composition languages should support the invocation of services through which software functionalities are exposed to application developers.

Although SOA provides good foundation for development and deployment of reusable and flexible business processes, to support event-driven process execution, service-oriented architecture is combined with event-driven architecture (EDA) [5-8]. EDA defines an architectural pattern for designing and implementing applications in which events transmit between decoupled software components.

While our SOA- and EDA-ready languages along with supporting service-oriented programming model and distributed language interpreters were presented in our previous papers [9-11, 14-18, 23], in this paper we discuss challenges that must be met while designing such languages. In a proposed language design methodology, we identify basic requirements that influence the language design decisions. The basic set of requirements includes language standardization, integration capabilities for heterogeneous systems and services, event-driven workflow support, and scripting-based simplicity. The requirements are derived from properties of typical applications implemented in today's information systems. Languages designed by proposed methodology were tested in a number of practical usage scenarios. As a use-case in this paper, we are using healthcare information system to show the applicability of proposed methodology in a language design for such a demanding environment.

The rest of the paper is organized as follows. In Section 2, we briefly describe our implementation of event-driven service-oriented architecture upon which event-driven service compositions are built. In Section 3, we describe a language design methodology where event-driven service composition languages are derived from standards-based SOA languages and widely used scripting and coordination languages. Section 4 gives a generic example of an application based on event-driven service composition that is used throughout the paper. In Section 5, we present *Coopetition services* as special-purpose services for handling events in service-oriented applications. In Section 6, we present standards-based service composition language CL, while its simplified and coordination-based counterpart language SSCL is described in Section 7. In Section 8, we present a multi-stage translation framework that enables translation of high-level SSCL programs into low-level CL processes and supports collaborative work of application developers with different knowledge and skills. In Section 9, we compare the efficiency of application development process using XML-based CL language and compact and textual SSCL language. Section 10 concludes the paper.

2 EVENT-DRIVEN SOA

Design of an event-driven service composition language begins with an analysis of the relationships between the language and the underlying software architecture. These relationships determine how events are handled within user programs, which in turn has an impact on design of the language primitives.

Since basic SOA does not support event-driven workflows, it is augmented with special-purpose mechanisms for handling events. While event-driven SOA platforms presented in [5-8] are using event-handling components integrated into the SOA middleware, we have exposed them to application developers as special-purpose event-handling services called *Coopetition services* (CS) [9-11]. This provides greater flexibility of the architecture and easier introduction of new event-handling mechanisms into the system. Furthermore, application developers are enabled to use services as basic building blocks for application-specific functionalities as well as for integration of services into event-driven workflows. This uniformity allows seamless and simple integration of event-driven mechanisms with application-level logic.

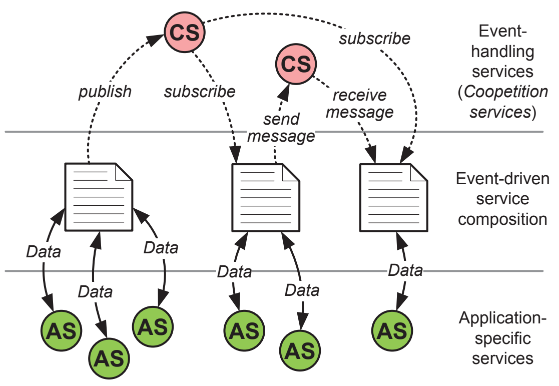


Fig. 1. Event-driven service-oriented architecture

Our implementation of event-driven SOA is shown in Fig. 1. A set of application-specific services (AS) is composed into business-specific workflow using service composition language. The interaction between user programs written in service composition language and application-specific services relies mostly on synchronous request-response paradigm. To augment SOA with event handling properties, we have developed special-purpose *Coopetition services* (CS) that implement fundamental EDA characteristics, such as decoupled interactions, many-to-many communication, publish/subscribe messaging, event triggering, and asynchronous operations. Since the complexity of event processing is hidden behind the *Coopetition services*, a service composition language is considered EDA-ready if it supports the invocation of event-handling services. Event-handling services are explained in details in Section 3, while design of language primitives for their invocation is presented in Sections 4 and 5.

3 LANGUAGE DESIGN METHODOLOGY

During design of a programming language for event-driven service composition, we analyzed the adoption of existing SOA-ready and general-purpose programming languages in different programmer communities. Our objective was not to design a new language from scratch, but rather augment most representative existing language with SOA&EDA-ready properties.

WS-BPEL [12, 13] and similar XML-based languages are standardized languages for development of SOA-based applications. On the other hand, wide population of software developers today is using scripting and coordination languages for rapid application development. Therefore, we designed two programming languages with distinct features. The influential languages that drove the design of our event-driven service composition languages and key features of each language are shown in Fig. 2. Languages are presented in details in Sections 4 and 5.

To stay aligned with standardization in service-oriented computing and enable data exchange in heterogeneous environment, we designed an XML-based service composition language named *Coopetition Language* (CL) [14-16]. CL is derived from WS-BPEL and WSDL languages. Since WS-BPEL and WSDL are standardized languages for building SOA-based applications, CL inherits the SOA properties from these languages. To be EDA-ready, CL is extended with invocations of event-handling services.

We simplified the XML-based syntax of CL language and made the programming more efficient and less error-prone by designing *Simple Service Composition Language* (SSCL) [17, 18]. SSCL is an application-level end-user language inspired by scripting and coordination languages. SSCL emphasizes application-specific properties of event-driven service composition and hides XML markup from application developers. To be SOA and EDA-ready, SSCL consists of two types of programming primitives. To be SOA-ready, language contains a generic primitive for invocation of Web services. This primitive is used to invoke application-specific services. To be EDA-ready, a set of special-purpose primitives is used to invoke event-handling services and to compose application-specific services into event-driven workflows.

4 EVENT-DRIVEN SERVICE COMPOSITION EXAMPLE

Contemporary complex cyber-physical, socio-technical, and bio-medical systems rely on information technology that integrates and coordinates various technical, financial, medical, biological, and social processes. Openness

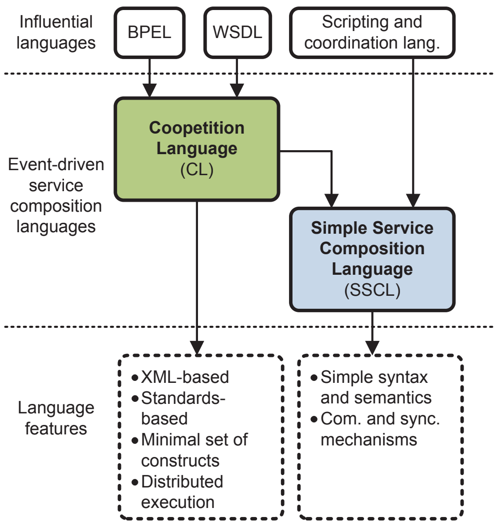


Fig. 2. Design of event-driven service composition languages

to technical, social, and life environments requires online, open-ended, interactive, concurrent, and event-driven information systems. As a result, the most suitable technology for development of such information systems is based on service-oriented event-driven architecture. Since healthcare application integrates cyber-physical, socio-technical, and bio-medical systems, we use it as an example to demonstrate the applicability of our methodology in design of SOA- and EDA-ready languages.

Healthcare systems are subject to different demographic, legal, medical, and administrative practices. Even basic business processes, such as electronic prescriptions, may widely vary from country to country or even within different administrative domains of the same country. Therefore, it becomes mandatory to build e-Health solutions that are reusable, flexible, and responsive. Figure 3 presents an example of event-driven service composition that implements e-Prescription process within a typical e-Health solution.

E-Prescription process begins with primary general practitioner (GP), who obtains patient's medical record from electronic healthcare record system (EHCR) (1). After obtaining patient's medical record and consulting with the patient, primary GP establishes the initial diagnosis and requests a second opinion from one of his colleague GPs (2). One of the colleague GPs examines the case and gives his own opinion back to primary GP (3). After establishing the final diagnosis, primary GP updates patient's EHCR and forwards the diagnosis to the Healthcare Control Body (HCCB) for inspection (4). HCCB is governmental agency that controls spending and utilization of healthcare resources within the healthcare system. HCCB inspects the diagnosis and associated prescription, and approves the issuance of prescribed medicine by sending a notification to all the drugstores (DS) where patient can obtain it (5).

The implementation of given e-Prescription process using event-driven service composition is presented in Fig. 4. E-Health subsystems, such as EHCR, GP application, HCCB, and DS are exposed as application-specific services. These services are interconnected using a set of coordination tasks that invoke application-specific services and coordinate themselves through event-handling mechanisms for event-triggered task execution, asynchronous communication, and publish/subscribe messaging. In the example shown in Fig. 4, we are using two different coordination mechanisms: message queue (*MQ1*, *MQ2*) for decoupled asynchronous communication and broker center (*BC1*, *BC2*) for publish/subscribe messaging. Task logic is expressed in a pseudo code resembling the constructs of a typical scripting language.

The presented implementation consists of five tasks. At the beginning, *Task 1* obtains patient identifier from GP

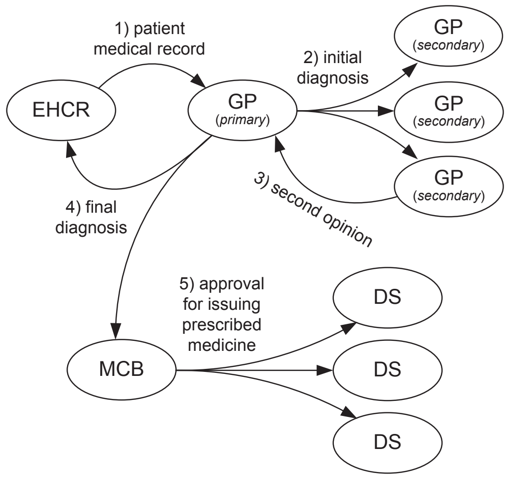


Fig. 3. Event-driven service composition for e-Prescription process

application, patient's healthcare record from EHCR, and initial diagnosis from primary GP, using generic *Invoke* primitive to invoke application-specific services. After obtaining the initial diagnosis, *Task 1* forwards the case to any secondary GP by storing it to message queue *MQ1* (*Send Message (MQ1, diag)*). Each secondary GP has associated *Task 2* that obtains the case stored in message queue *MQ1* (*Receive Message (MQ1)*) once the respective secondary GP becomes available, forwards the case to the secondary GP for diagnosis (*Invoke (GPx, getAltDiag, diag)*), and returns the alternative diagnosis back to primary GP through *MQ2* (*Send Message (MQ2, altDiag)*). Once the alternative diagnosis becomes available, *Task 1* continues with the execution. It obtains the alternative diagnosis from *MQ2* (*Receive Message (MQ2)*), invoke the primary GP to get the final diagnosis (*Invoke (GP1, getFinDiag, altDiag)*), and announces this diagnosis through the publish/subscribe mechanism *BC1* (*Publish (BC1, patID, finDiag)*).

Task 3 and *Task 4* are subscribed to receive announcements published on *BC1* (*Subscribe (BC1, patID, finDiag, INT1)*). As part of the subscription, subscribing tasks provide the address of the interpreter service (*INT1*) responsible for matching published events with the subscriptions. The details about three-tiered publish/subscribe system that consists of publishers and subscribers as standard elements, and interpreters as our extension of standard publish/subscribe system, are given in Section 3. Each time an announcement occurs, the announced event is forwarded to the interpreter service. If there is a match, *Task 3* and *Task 4* start the associated event handlers. *Task 3* invokes the EHCR to update patient's healthcare record with newly established diagnosis (*Invoke (EHCR, updateHCR, patID,* *finDiag))*. At the same time, *Task 4* invokes the HCCB to get the approval for issuing the prescribed medicine to the patient (*Invoke (HCCB, getApproval, patID, finDiag)*). The acknowledgment retrieved from HCCB is then broadcasted to all the drugstores in the system through publish/subscribe mechanism *BC2* (*Publish (BC2, ack)*). Each drugstore service has associated *Task 5* that is subscribed to receive announcements from *BC2* (*Subscribe (BC2, ack, INT2)*) with *INT2* as interpreter service. *Task 5* invokes the associated drugstore service each time such announcement occurs (*Invoke (DSx, setAck, ack)*).

5 COOPETITION SERVICES

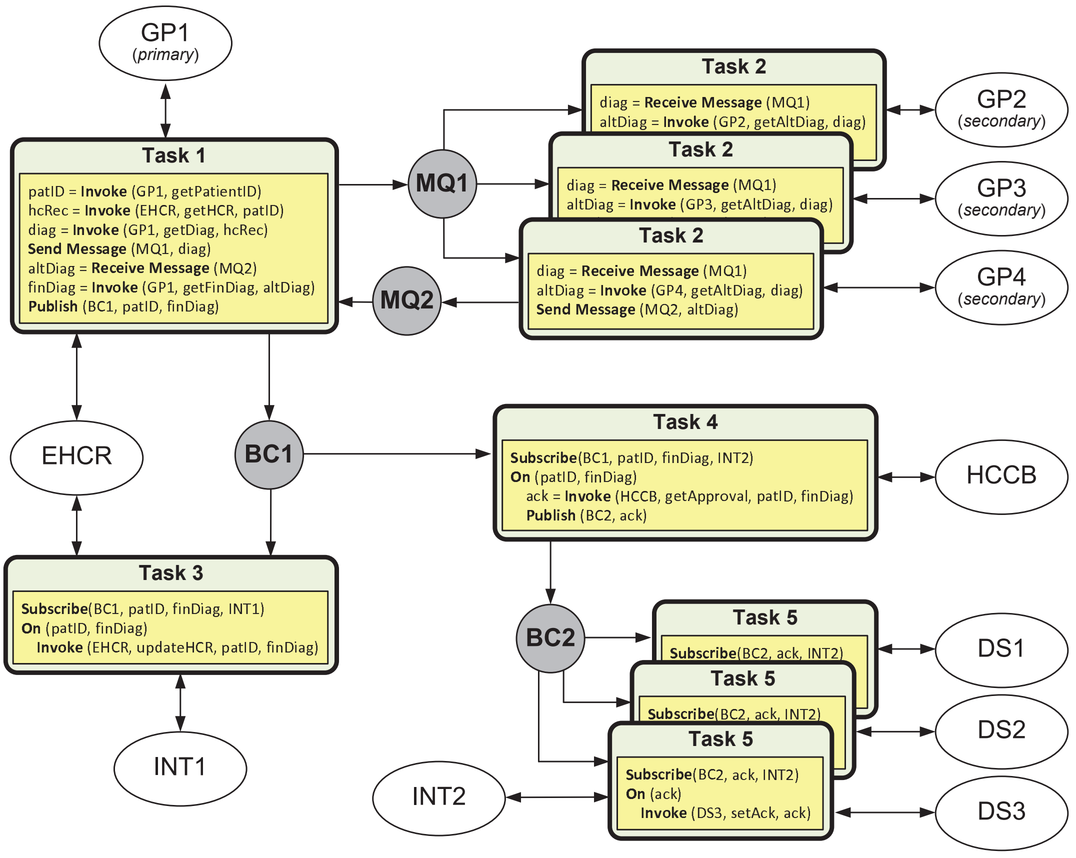
To compose services into event-driven service compositions, we need special-purpose mechanisms for handling events in distributed and heterogeneous environment. These mechanisms implement fundamental features of event-driven architecture (EDA), such as decoupled interactions, event-triggering, asynchronous communication, many-to-many communication, and publish/subscribe messaging.

As fundamental components of event-driven SOA, event-handling mechanisms should also be implemented as services. Therefore, we developed a generic set of event-handling mechanisms and expose them as *Coopetition services* (cooperation + competition) [9-11]. *Coopetition services* provide common set of event-driven interaction patterns in distributed systems [17, 18]. Programming languages presented in the rest of this paper use these services for implementation of event-driven service-oriented workflows.

Queue is a messaging service that implements FCFS communication pattern. Although primarily designed for decoupled one-to-one communication, it can be used for different forms of one-to-many and many-to-many communication. There are two modes of operation of the *Queue* service: blocking and non-blocking mode. In blocking mode, the task reading a message from an empty queue remains blocked until at least one message becomes available. In non-blocking mode, the task continues its execution despite the empty queue and gets notified to continue with the communication when the message becomes available. While non-blocking mode enables implementation of asynchronous communication of service composition tasks, blocking mode is used for implementation of event-triggered communication. Putting a message to a queue corresponds to raising an event, while reading a message from a queue corresponds to consuming the event. Furthermore, since each message sent to the queue contains a piece of information, the *Queue* service may be used for implementation of simple publish/subscribe system and different forms of one-to-many and many-to-many communication.

To enable more user-friendly implementation of event-triggering system that involves multiple parties, we designed the *TokenCenter* service. The *TokenCenter* service implements counting semaphore for mutual exclusion and synchronization of two or more concurrent tasks. Event-triggered execution of service composition tasks is achieved through token passing. Putting tokens to a token center corresponds to raising an event, while retrieving tokens from a token center corresponds to consuming the event. In contrast to the *Queue* where messages are sent and retrieved from the queue one at a time, *TokenCenter* allows multiple tokens to be acquired or returned in a single transaction. This allows better control over the system when multiple tasks are competing for the limited number of shared resources. There are two modes of operation of the *TokenCenter* service: blocking and non-blocking mode. In blocking mode, the task requesting for tokens from an empty token center or token center with insufficient number of tokens remains blocked until at least the requested number of tokens becomes available. In non-blocking mode, the task continues its execution despite the insufficient number of tokens in the token center and gets notified to continue with the critical section when the requested number of tokens becomes available.

Since tokens and messages are delivered to the tasks on a first-come first-served basis regardless of the information contained within the message, both *Queue* and *TokenCenter* are limited to the serialized event triggering. To enable out-of-order content-based event triggering, we designed the *BrokerCenter* service. The *BrokerCenter* service enables event-triggered execution of service composition tasks by using publish/subscribe messaging. We extended the basic two-tiered publish/subscribe model that consists of publishers and subscribers to three-tiered *publish/subscribe/interpret* model that consists of publishers, subscribers, and interpreters. First two parties, publishers and subscribers, have the same roles as in regular publish/subscribe model. Publishers announce information, while subscribers register terms of interest to the *BrokerCenter*. To keep the *BrokerCenter* application-independent service, the third parties included into the model are interpreters that interpret published events in order to match them with the subscriptions and notify the subscribers. As part of the subscription process, subscribers provide the *BrokerCenter* with address of the interpreter service responsible for interpretation of the events they are interested in. Interpreter services analyze information announced by publishers according to the subscribed terms. If the announced information satisfies the terms, the interpreter service triggers the subscriber by sending a notification. For example, in an e-Prescription application, the terms could specify that only positive acknowledgments from HCCB are accepted by drugstore services. Terms,



```
Task 1
patID = Invoke (GP1, getPatientID)
hcRec = Invoke (EHCR, getHCR, patID)
diag = Invoke (GP1, getDiag, hcRec)
Send Message (MQ1, diag)
altDiag = Receive Message (MQ2)
finDiag = Invoke (GP1, getFinDiag, altDiag)
Publish (BC1, patID, finDiag)
```

```
Task 2
diag = Receive Message (MQ1)
altDiag = Invoke (GP2, getAltDiag, diag)
```

```
Task 2
diag = Receive Message (MQ1)
altDiag = Invoke (GP3, getAltDiag, diag)
```

```
Task 2
diag = Receive Message (MQ1)
altDiag = Invoke (GP4, getAltDiag, diag)
Send Message (MQ2, altDiag)
```

```
Task 4
Subscribe(BC1, patID, finDiag, INT2)
On (patID, finDiag)
   ack = Invoke (HCCB, getApproval, patID, finDiag)
   Publish (BC2, ack)
```

```
Task 3
Subscribe(BC1, patID, finDiag, INT1)
On (patID, finDiag)
   Invoke (EHCR, updateHCR, patID, finDiag)
```

```
Task 5
Subscribe(BC2, ack, INT2)
```

```
Task 5
Subscribe(BC2, ack, INT2)
```

```
Task 5
Subscribe(BC2, ack, INT2)
On (ack)
   Invoke (DS3, setAck, ack)
```

Fig. 4. Implementation of event-driven e-Prescription process

announcements, and notifications are user-defined documents. Having the interpretation of the event implemented as a separate interpreter service, rather than being an integral part of the application, provides great flexibility in designing publish/subscribe systems. Interpreters are independent services developed by independent developers. Different applications may use different interpreters, the same interpreter may be used in multiple applications, and multiple interpreters may be simultaneously used within single application. Furthermore, the interpreters assigned to particular subscriptions may dynamically change during the run-time.

6 COOPETITION LANGUAGE (CL)

To make the development of event-driven applications familiar to service-oriented programmer community, we designed an event-driven service composition language derived from XML-based service composition languages. Since WS-BPEL [12] is widely accepted as standard executable language in SOA-based applications, our event-driven service composition language is based on WS-BPEL. The new SOA&EDA-ready language is called *Coopetition Language* (CL) [14-16].

Standardized XML-based languages, such as WS-BPEL [12] and WSDL [13], enable technology-transparent process descriptions in hybrid environments comprised of heterogeneous components. Since they were designed for implementation of internet-based applications, they support advanced concepts like task-level concurrency, transactions, and complex data integration. Furthermore, a lot of interpreters and execution frameworks for these languages already exist on the market. Our goal was to stay compliant with existing XML-based languages, reuse the expressiveness of standard WS-BPEL and WSDL languages, and augment them with support for event-driven workflows.

Coopetition Language (CL) [14-16] reuses basic WS-BPEL constructs. Since WS-BPEL already supports the invocation of Web services, the CL language inherits SOA-ready properties from WS-BPEL. However, to make the CL an EDA-ready service composition language, we augment the WS-BPEL process descriptions with WSDL descriptions of *Coopetition services*. Being an integral part of the CL language, WSDL descriptions of *Coopetition ser-*

vices enable the invocation of event handling mechanisms directly from CL programs.

Figure 5 shows the implementation of *Task 1* from Fig. 4 in CL language. Since both application-specific functionalities and event-handling mechanisms are exposed as services, the CL program consists of a sequence of *invoke* statements, which is a standard WS-BPEL construct for invoking Web services. First three statements are used to invoke the *getPatientID*, *getHCR*, and *getDiag* operations of application-specific services *GP1*, *EHCR*, and *GP1*, respectively. Next two statements are used to send a message to *Task 2* through *Queue* service *MQ1* and read a response from *Task 2* through *Queue* service *MQ2*. Finally, the program invokes application-specific service *GP1* and publishes the results of its execution to the *BrokerCenter* service *BC1*.

```
<sequence>
  <invoke partnerLink="GP1"
          operation="getPatientID"
          portType="GP"
          requestVariable="patIDReq"
          responseVariable="patID"/>

  <invoke partnerLink="EHCR"
          operation="getHCR"
          portType="EHCR"
          requestVariable="patID"
          responseVariable="hcRec"/>

  <invoke partnerLink="GP1"
          operation="getDiag"
          portType="GP"
          requestVariable="hcRec"
          responseVariable="diag"/>

  <invoke partnerLink="MQ1"
          operation="Put"
          portType="MessageQueue"
          requestVariable="diag"/>

  <invoke partnerLink="MQ2"
          operation="Get"
          portType="MessageQueue"
          requestVariable="msgReq"
          responseVariable="altDiag"/>

  <invoke partnerLink="GP1"
          operation="getFinDiag"
          portType="GP"
          requestVariable="altDiag"
          responseVariable="finDiag"/>

  <invoke partnerLink="BC1"
          operation="Publish"
          portType="BrokerCenter"
          requestVariable="pat_diag"/>
</sequence>
```

Fig. 5. CL implementation of Task 1

7 SIMPLE SERVICE COMPOSITION LANGUAGE (SSCL)

Building XML-based process descriptions requires the use of complex lexical and syntax patterns. XML markup becomes unpractical for complex processes since their descriptions become too large and unmanageable. The usual way to manage complex XML syntax while building service compositions are visual editors, such as BPEL Project for Eclipse [19] and BPEL Editor for the .NET Framework [20]. These tools provide a form-based GUI for definition of application-specific XML parameters and automatically generate XML markup on behalf of application developer.

Instead of using form-based XML editors, we designed text-based language called *Simple Service Composition Language* (SSCL) [17, 18]. SSCL was inspired by scripting and coordination languages which, when specialized for given domain, enable rapid and simple application development. On one side, the objective of the SSCL language is to make the programming more efficient and less error-prone. On the other side, the objective is to design a language that resembles the process where set of concurrent tasks is mutually coordinated through event-handling EDA mechanisms.

During the design of the SSCL language, we reused the XML-based CL language constructs and designed their textual counterparts. SSCL notation is based on simple human-readable lexical and syntax features with statements that have simple semantics and usage patterns.

Figure 6 shows the implementation of *Task 1* from Fig. 4 in SSCL language. SOA and EDA-ready properties of SSCL language are accomplished through two types of language primitives. To be SOA-ready, SSCL contains a generic *invoke* statement for invocation of application-specific services. To be EDA-ready, a set of special-purpose statements for invocation of *Coopetition services* is designed. For example, *putmessage* and *getmessage* statements are used to handle messages stored in the *Queue*, while *publish* statement is used to publish events to the *BrokerCenter*.

8 SSCL-TO-CL TRANSLATION AND EXECUTION FRAMEWORK

SSCL applications are executed in distributed environment by translating SSCL programs to CL code [14, 17]. Figure 7 presents the basic elements of SSCL-to-CL translation and execution environment.

SSCL programs that make event-driven SOA applications are translated by specialized *SSCL Compiler*. The *SSCL Compiler* uses *Service description repository* to fetch WSDL interface descriptions of each service invoked in SSCL programs. The logic of SSCL programs is translated using predefined template code snippets written in CL language and stored in *Snippets repository*. Once CL snippets are filled-in and merged into complete CL programs, the resulting CL programs are deployed and executed by *CL Interpreter*. *CL Interpreter* is a lightweight WS-BPEL-based engine that supports execution of CL programs. During their execution, CL programs invoke *Application-specific services* that perform application-specific computations and *Coopetition services* that handle events.

Translation and execution environment for SSCL and CL languages has been built using PIE (*Programmable Internet Environment*, http://www.pie.fer.hr). PIE is a distributed service-oriented platform for deployment and execution of Web services. PIE has been developed as a part of the *CroGrid* national poly-project supported by the Ministry of Science, Education, and Sports of the Republic of Croatia in cooperation with Ericsson Nikola Tesla, Zagreb, Croatia.

9 EVALUATION OF CL AND SSCL

To compare the efficiency of application development process using XML-based CL language and compact and

```
program Task1
 variable patID, hcRec, diag, altDiag,
          finDiag
 invoke "http://eh.com/GP1",
        "getPatID", patID
 invoke "http://eh.com/EHCR",
        "getHCR", patID, hcRec
 invoke "http://eh.com/GP1",
        "getDiag", hcRec, diag
 putmessage "http://eh.com/Queue",
            "MQ1", diag
 getmessage "http://eh.com/Queue",
            "MQ2", altDiag
 invoke "http://eh.com/GP1",
        "getFinDiag", altDiag, finDiag
 publish "http://eh.com/BrokerCenter",
         "BC1", "permanent", patID,
         finDiag, eventHandleID
endprogram
```

Fig. 6. SSCL implementation of Task 1

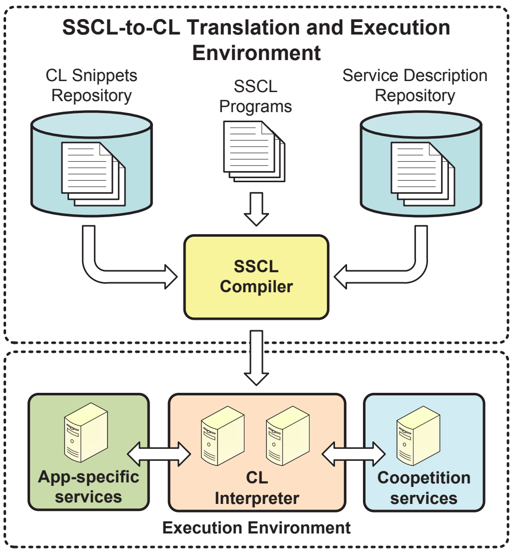


Fig. 7. SSCL translation and execution

textual SSCL language, we made a series of experiments with computer science PhD students from University of Zagreb attending the Ericsson Nikola Tesla summer camp. Based on given set of services and service composition workflows, students were required to apply the languages to compose these services into applications for distributed algebraic computations, as well as simple e-Health, financial, and GIS applications [21, 22]. We evaluated the two languages against two criteria: *code size reduction* and *application development time reduction* [23]. The results of experiments are summarized in Table 1.

Table 1. Evaluation of CL and SSCL

		CL	SSCL
Code Size		5000 – 10000 lines of code	30 – 50 lines of code
Development Effort	***Man Power***	3 – 5 developers	1 developer
	Time	5 – 10 days	2 days

As Table 1 shows, typical application based on event-driven service composition requires several thousands of lines of code if CL language is used. Relatively large amount of code is required due to extensive XML markup used in CL language and WSDL descriptions of application-specific and *Coopetition services* embedded into the CL programs. On the other hand, an equivalent application written in SSCL requires only few dozens of lines of code. Therefore, SSCL reduces the amount of code for two orders of magnitude, making the application maintenance far easier.

Second criterion we used during the language evaluation is *application development time reduction*. As Table 1 shows, typical application based on event-driven service composition written in CL requires approximately 30 man-days of development effort. On the other hand, an equivalent application written in SSCL requires approximately two man-days. Therefore, SSCL reduces the application development time by at least an order of magnitude.

10 CONCLUSION

This paper describes methodology for design of SOA- and EDA-ready programming languages. In our language design methodology, we identify four main requirements. First, a language should be based on standard service composition languages to be compatible with SOA-enabling technologies. Second, a language should enable seamless integration of services into event-driven workflows. Third, a language should resemble scripting and coordination languages, which are today widely used in application/software development. Fourth, to overcome a knowledge divide, language should enable seamless cooperation between application developers with different skills and knowledge.

Since it is hard to satisfy all design requirements in a single language, our methodology span two different abstraction levels, ranging from system level XML languages to application level coordination languages, which targets two different groups of application developers with different skills and knowledge, ranging from professional programmers familiar with XML-based service composition technologies to end-users and business analysts familiar with business processes.

To stay aligned with SOA community and Web services, we choose standardized XML-based WS-BPEL language as a basis for new event-driven service composition language. Since basic SOA does not support event-driven workflows, we upgraded the WS-BPEL with *support for event-driven application design*. We have designed special-purpose *Coopetition services* for event triggering, decoupled interactions, synchronization, many-to-many communication, publish/subscribe messaging, and asynchronous operations. We augmented the WS-BPEL process descriptions with WSDL descriptions of *Coopetition services*. Being an integral part of a newly developed CL language (*Coopetition Language*), WSDL descriptions of *Coopetition services* enable the invocation of event-handling mechanisms directly from the CL programs.

While upgraded WS-BPEL is convenient for use by a community of professional programmers familiar with service-oriented architecture and Web services, end-users and business analysts familiar with business processes still find this language complex and intractable. They prefer high-level descriptions of business processes by using scripting and coordination languages. Therefore, we designed *textual coordination language* SSCL (*Simple Service Composition Language*) with simple syntax structure and convenient semantics and usage patterns. The core elements of SSCL are programming primitives for invocation of Web services and handling events through *Coopetition services*.

We found the SSCL language more productive for business process designers than WS-BPEL based CL language. During summer internships in Ericsson Nikola Tesla Zagreb, we examined the productivity of SSCL and CL languages within different groups of students. Our experience shows that SSCL reduces the code size and application development time by one to two orders of magnitude if compared to WS-BPEL-based CL language.

To enable a seamless cooperation between application developers with different skills and knowledge, we defined a multistage process for translation of service composition logic from high to low level of abstraction. This process translates the SSCL language into the CL language, which is then executed by WS-BPEL interpreter augmented with WSDL descriptions of *Coopetition services*. Multistage translation enables cooperation between end-users and professional programmers. For example, an end-user who understands the logic of business process may define the core application logic in the high-level SSCL language. After being translated into the low-level and more expressive CL language, professional programmers may augment the core process with additional logic, such as conversion and adaptation of service parameter data formats, which is required for correct execution of service compositions.

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, Miroslav Popovic, Klemo Vladimir, Marin Silic, Goran Delac, Jakov Krolo, Ivan Budiselic, and Josko Radej from School of Electrical Engineering and Computing, University of Zagreb, 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 PIE framework.

APPENDIX A

Coopetition service API

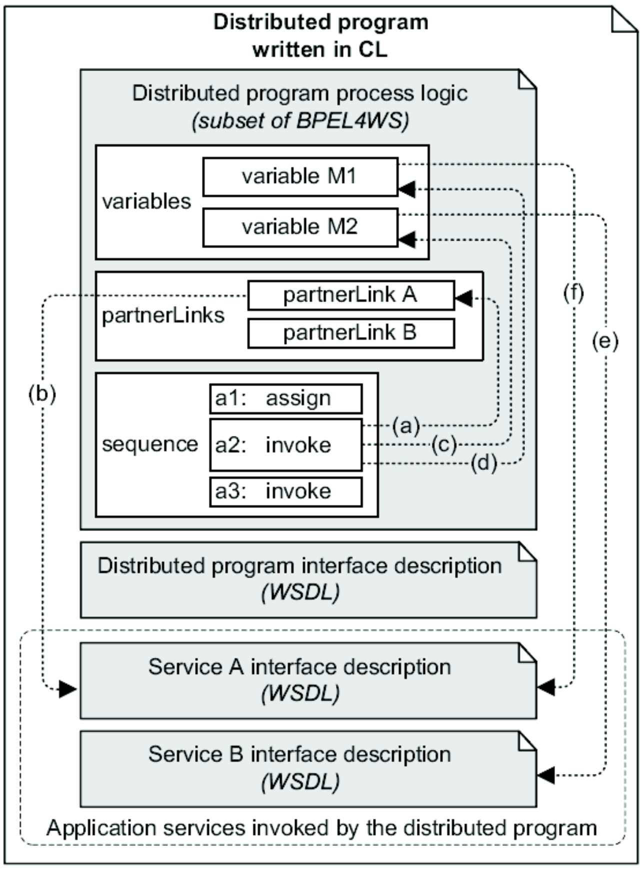
Service	Service API	Description
Binary Semaphore (token center with exactly one token)	**Create**(*bsemURL*, *instID*)	Creates an instance of a binary semaphore
	Destroy(*bsemURL*, *instID*)	Destroys an instance of a binary semaphore
	Obtain(*bsemURL*, *instID*)	Obtains token from a binary semaphore
	Release(*bsemURL*, *instID*)	Returns token to a binary semaphore
Counting Semaphore (token center with arbitrary number of tokens)	**Create**(*csemURL*, *instID*, *capacity*)	Creates an instance of a counting semaphore with *capacity* tokens
	Destroy(*csemURL*, *instID*)	Destroys an instance of a counting semaphore
	Obtain(*csemURL*, *instID*, *N*)	Obtains *N* tokens from a counting semaphore
	Release(*csemURL*, *instID*, *N*)	Returns *N* tokens to a counting semaphore
Message Queue	**Create**(*mqURL*, *instID*)	Creates an instance of a message queue
	Destroy(*mqURL*, *instID*)	Destroys an instance of a message queue
	Put(*mqURL*, *instID*, *msg*)	Sends a message to a message queue
	Get(*mqURL*, *instID*)	Retrieves a message from a message queue
Event Channel (broker center)	**Create**(*ecURL*, *instID*)	Creates an instance of an event channel
	Destroy(*ecURL*, *instID*)	Destroys an instance of an event channel
	Publish(*ecURL*, *instID*, *eventType*, *eventDoc*)	Publishes an event to an event channel
	Republish(*ecURL*, *instID*, *eventType*, *eventID*, *eventDoc*)	Republishes an event to an event channel
	Unpublish(*ecURL*, *instID*, *eventID*)	Revokes the event published to an event channel
	Subscribe(*ecURL*, *instID*, *interpreterURL*, *callback*, *subscriptionDoc*)	Starts listening for events published to an event channel
	Unsubscribe(*ecURL*, *instID*, *subscriptionID*)	Cancels an active subscription with an event channel

SSCL language primitives

Language primitive	Description
createBinarySemaphore *bsemURL*, *instID*	Creates an instance of a binary semaphore
destroyBinarySemaphore *bsemURL*, *instID*	Destroys an instance of a binary semaphore
obtainBinarySemaphore *bsemURL*, *instID*	Obtains token from a binary semaphore
releaseBinarySemaphore *bsemURL*, *instID*	Returns token to a binary semaphore
createCountingSemaphore *csemURL*, *instID*, *capacity*	Creates an instance of a counting semaphore with *capacity* tokens
destroyCountingSemaphore *csemURL*, *instID*	Destroys an instance of a counting semaphore
obtainCountingSemaphore *csemURL*, *instID*, *N*	Obtains *N* tokens from a counting semaphore
releaseCountingSemaphore *csemURL*, *instID*, *N*	Returns *N* tokens to a counting semaphore

createMailbox *mqURL, instID*	Creates an instance of a message queue
destroyMailbox *mqURL, instID*	Destroys an instance of a message queue
putMessage *mqURL, instID, msg*	Sends a message to a message queue
getMessage *mqURL, instID, msg*	Retrieves a message from a message queue
createEventChannel *ecURL, instID*	Creates an instance of an event channel
destroyEventChannel *ecURL, instID*	Destroys an instance of an event channel
publish *ecURL, instID, eventType, eventDoc, eventID*	Publishes an event to an event channel
republish *ecURL, instID, eventType, eventID, eventDoc, newEventID*	Republishes an event to an event channel
unpublish *ecURL, instID, eventID, result*	Revokes the event published to an event channel
subscribe *ecURL, instID, interpreterURL, callback, subscriptionDoc, subscriptionID*	Starts listening for events published to an event channel
unsubscribe *ecURL, instID, subscriptionID, result*	Cancels an active subscription with an event channel
invoke *serviceURL, operationName, parameterList*	Invokes a stateless application-specific service
invoke *serviceURL,* instID, *operationName, parameterList*	Invokes an instance of a stateful application-specific service

Data structures of the CL programs


Distributed program written in CL
Distributed program process logic
(subset of BPEL4WS)
variables
variable M1
variable M2
partnerLinks
partnerLink A
partnerLink B
sequence
a1: assign
a2: invoke
a3: invoke
(a)
(b)
(c)
(d)
(e)
(f)
Distributed program interface description
(WSDL)
Service A interface description
(WSDL)
Service B interface description
(WSDL)
Application services invoked by the distributed program

An excerpt of CL language primitives
(*message queue* coopetition service + generic application-specific service)

Language primitive	Description

`<invoke partnerLink ="messageQueuePartnerLinkRef"`
`Operation ="Create"`
`portType ="messageQueuePortTypeRef"`
`requestVariable ="mqInstanceNameVarRef"`
`responseVariable ="mqCreateResVarRef" />`
	Creates an instance of a message queue

`<invoke partnerLink ="messageQueuePartnerLinkRef"`
`Operation ="Destroy"`
`portType ="messageQueuePortTypeRef"`
`requestVariable ="messageQueueInstanceRef"`
`responseVariable ="mqDestroyResVarRef" />`
	Destroys an instance of a message queue

`<invoke partnerLink ="messageQueuePartnerLinkRef"`
`Operation ="Get"`
`portType ="messageQueuePortTypeRef"`
`requestVariable ="messageQueueInstanceRef"`
`responseVariable ="outputMsgVarRef" />`
	Retrieves a message from a message queue

`<invoke partnerLink ="messageQueuePartnerLinkRef"`
`Operation ="Put"`
`portType ="messageQueuePortTypeRef"`
`requestVariable ="messageQueueInstanceRef" />`
	Sends a message to a message queue

`<invoke partnerLink ="servicePartnerLinkRef"`
`Operation ="serviceOperationRef"`
`portType ="servicePortTypeRef"`
`requestVariable ="inputParamsVarRef"`
`responseVariable ="outputDataVarRef" />`
	Generic primitive for invocation of an application-specific service

REFERENCES

[1] R. W. Schulte, "The Growing Role of Events in Enterprise Applications", *Gartner Research*, July 2003, http://www.gartner.com/resources/116100/116129/116129.pdf, accessed 17 Sep 2010.

[2] Z. Laliwala, S. Chaudhary, "Event-driven Service-Oriented Architecture", *International Conference on Service Systems and Service Management*, pp. 1-6, Melbourne, VIC, Australia, July 2008.

[3] "Event-Driven SOA: A Better Way to SOA", *TIBCO Software Inc.*, 2006, http://www.tibco.com/multimedia/wp-event-driven-soa_tcm8-803.pdf, accessed 14 Dec 2010.

[4] N. Milanovic, "Service Engineering Design Patterns", *Proceedings of the 2nd IEEE International Symposium on Service Oriented System Engineering*, pp. 19-26, Shanghai, China, 2006.

[5] J.-L. Marechaux, "Combining Service-Oriented Architecture and Event-Driven Architecture using an Enterprise Service Bus", *IBM DeveloperWorks*, March 2006, http://www.ibm.com/developerworks/library/ws-soa-eda-esb/, accessed 14 Dec 2010

[6] place 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, 2005, pp. 158-166

[7] "Service Component Architecture – Unifying SOA and EDA", Whitepaper, *Fiorano Software Technologies*, 2010, http://www.fiorano.com/products/bca_overview.php, accessed 14 Dec 2010.

[8] J. Hanson, "Event-Driven Services in SOA: Design an Event-Driven and Service-Oriented Platform with Mule", *JavaWorld.com*, January 2005, http://www.javaworld.com/javaworld/jw-01-2005/jw-0131-soa.html, accessed 14 Dec 2010.

[9] A. Milanovic, S. Srbljic, D. Skrobo, D. Capalija, S. Reskovic, "Coopetition Mechanisms for Service-Oriented Distributed Systems", *Proceedings of 3rd International Conference on Computing, Communications and Control Technologies, Cybernetics and Informatics (CCCT'05)*, pp. 118–123, Austin, Texas, USA, July 2005.

[10] S. Reskovic, *Synchronization and Communication Mechanisms for Service-Oriented Applications*, B.Sc. Thesis, School of EE & Computing, University of Zagreb, September 2005 (in Croatian).

[11] D. Capalija, *Publish/Subscribe Mechanisms for Implementation of Content-Based Networking*, B.Sc. Thesis, School of EE & Computing, University of Zagreb, June 2005 (in Croatian).

[12] A. Alves, et al., *Web Services Business Process Execution Language (WS-BPEL) 2.0*, OASIS, August 2006, http://www.oasis-open.org/committees/wsbpel, accessed 12 Feb 2010.

[13] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, *Web Services Decription Language (WSDL) 1.1*, World Wide Web Consortium (W3C) Note, February 2001, http://www.w3.org/TR/wsdl, accessed 12 Feb 2010.

[14] D. Skrobo, A. Milanovic, S. Srbljic, "Distributed Program Interpretation in Service-Oriented Distributed Systems", *Proceedings of the WMSCI'05*, pp. 193-197, Orlando, Florida, USA, July 2005.

[15] A. Milanovic, *Service-Oriented Programming Model*, Ph.D. Thesis, School of EE & Computing, University of Zagreb, December 2005 (in Croatian).

[16] D. Skrobo, *Distributed Program Interpretation in Service Oriented Architectures*, M.Sc. Thesis, School of EE & Computing, University of Zagreb, January 2006 (in Croatian).

[17] I. Gavran, A. Milanovic, S. Srbljic, "End-User Programming Language for Service-Oriented Integration", *Proceedings of 7th Workshop on Distributed Data and Structures*, Santa Clara, California, USA, January 2006.

[18] I. Gavran, *End-User Language for Service-Oriented Programming Model*, M.Sc. Thesis, School of EE & Computing,University of Zagreb, March 2006 (in Croatian).

[19] *BPEL Project*, http://www.eclipse.org/bpel, accessed 12 Feb 2010.

[20] M. Buckle, C. Abela, M. Montebello, "A BPEL Engine and Editor for the .NET Framework", *Proceedings of the 3rd IEEE European Conference on Web Services (ECOWS 2005)*, Vaxjo, Sweden, November 2005.

[21] I. Krka, *Personalized Service-Oriented Navigation System,* B.Sc. Thesis, School of EE & Computing, University of Zagreb, May 2007 (in Croatian).

[22] M. Zulj, *Service-Oriented Application for Accounting Management*, B.Sc. Thesis, School of EE & Computing, University of Zagreb, April 2007 (in Croatian).

[23] S. Srbljic, "PIE: End-user Programmable Internet Environment", *Google Technical Talk*, Mountain View, California, USA, January 2006.

Siniša Srbljić is currently a professor at the School of Electrical Engineering and Computing, University of Zagreb, and head of the Consumer Computing Laboratory. His career also spans Silicon Valley where he worked on large-scale distributed systems at AT&T Labs. He was visiting the University of Toronto, where he worked on the NUMAchine multiprocessor project, and the University of California, Irvine. His research interests include consumer computing and widget-oriented architecture. In teaching, he is involved in the theory of computing, programming language translation, service-oriented computing, and network middleware systems.

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 serviceoriented architectures, programming language design, end-user development, and consumer programming.

Daniel Skrobo is a solution architect at Ericsson Nikola Tesla d.d., Zagreb, Croatia. He received his Ph.D., M.Sc., and B.Sc. degrees from School of Electrical Engineering and Computing, University of Zagreb. Currently he is working on design and development of healthcare applications and systems. He held research assistantship position at School of Electrical Engineering and Computing, University of Zagreb and was a research engineering intern at Google's Mountain View office in CA, USA. His engineering and research interests are program translation systems and service-oriented computing systems.

AUTHORS' ADDRESSES

Prof. Siniša Srbljić, Ph.D.
Dejan Škvorc, Ph.D.
School of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, 10000, Zagreb, Croatia
email: sinisa.srbljic@fer.hr, dejan.skvorc@fer.hr

Daniel Skrobo, Ph.D.
Ericsson Nikola Tesla d.d.
Krapinska 45, p.p. 93, 10002, Zagreb, Croatia
email: daniel.skrobo@ericsson.com

Received: 2010-10-07
Accepted: 2011-01-24