Introducing domain-specific language implementation using web service-oriented technologies

Shih-Hsi Liu\textsuperscript{a,\ast }, Adam Cardenas\textsuperscript{a}, Marjan Mernik\textsuperscript{b}, Barrett R. Bryant\textsuperscript{c}, Jeff Gray\textsuperscript{d} and Xang Xiong\textsuperscript{a}

\textsuperscript{a}Department of Computer Science, California State University, Fresno, CA, USA
\textsuperscript{b}Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia
\textsuperscript{c}Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL, USA
\textsuperscript{d}Department of Computer Science, University of Alabama, Tuscaloosa, AL, USA

Abstract. Several advantages have been documented that suggest Domain-Specific Languages (DSLs) have the potential to improve productivity, reliability, maintainability and portability in some specialized domains. However, several key challenges still remain. In particular, the extension and evolution of both DSL syntax and semantics still suffer due to the limitations related to the current state-of-the-art implementation techniques. Such techniques also lack interoperable capabilities among base languages and limited tool support. As changes of domain concepts are omnipresent and more base languages may support DSL implementation, the aforementioned limitations may be no longer tolerable, and hence a new implementation technique to DSL development is desired. This paper implements six DSL case studies (representing imperative, declarative and hybrid categories) to validate the feasibility of utilizing Service-Oriented Architecture (SOA) for DSL implementation. Such case studies also highlight that the advantages of SOA (i.e., ease of evolution/extension, interoperability and tool support) can be retained under the context of DSL development. The paper concludes with the discussion of additional findings, both positive and negative: the SOA-based approach improves modularization at the lexical, syntactical and semantic levels and delegates tokenization/parsing to the underlying WS-BPEL engine; yet, the usability, resource utilization, security, and flexibility of the SOA-based DSLs are degraded, which requires more future work in this unique area that spans SOA and DSLs.

Keywords: Domain-specific languages, service-oriented architecture

1. Introduction

A Domain-Specific Language (DSL) \cite{8,37} is a programming or modeling language that introduces domain-specific notations at a higher level of abstraction. It has been demonstrated in many case studies that DSLs have the potential to improve productivity (5\text{–}10 times improvement), reliability, maintainability, and portability \cite{18}, especially in cases where end-users with a specific domain of experience are enabled to leverage their expertise to express some computational need \cite{37}.

There are many other additional advantages that support the adoption of a DSL that is tailored to a specific application domain. For example, DSLs offer better expressiveness \cite{37}, ease of use \cite{37}.
and have been demonstrated to be more accurate and efficient in program comprehension than General-Purpose Languages (GPLs) [27] and application libraries [26]. However, even with such benefits and success, implementing a DSL still requires much effort amid numerous challenges [37]. For example, Gray et al. [18] pointed out that DSLs might be unstable in their early development stage. Namely, a DSL may need to evolve both syntactically and semantically due to frequent needs to represent changes in the understanding of domain concepts. Additionally, poor interoperability with other languages and tool support are also looming concerns for those considering DSL adoption [18]. Kosar et al. [24] also investigated various implementation approaches to support DSL development. They found that DSL implementation using an interpreter or compiler pattern, which is detailed in Section 2.1, might suffer from extension/evolution problems. For example, DSL extension/evolution following the interpreter or compiler pattern requests the introduction or revision of associated language constructs. Also, their change consistencies among lexical, syntactical and semantic levels and implementation conformance to the underlying base language or compiler-generators need to be addressed. Therefore, a primary aim and motivation of this paper is to look for feasible techniques outside of the traditional language-implementation space that offer some promising results in support of DSL implementation, especially for addressing evolution/extension problems.

Service-Oriented Architecture (SOA) [12] is an architectural principle that composes a software system from coarse-grained and loosely coupled services. Such features facilitate software extension and evolution when business requirements change. A set of technology-neutral specifications and standards (e.g., WSDL [5] and WS-BPEL [40]) can assist with interoperability and integration. Additionally, many vendors have provided rich SOA development tool support, including Oracle Business Process Manager (BPM), Microsoft Visual Studio, Eclipse, and NetBeans, among others. Because of the aforementioned commonly known SOA benefits, this paper explores the implementation of six DSLs to validate the feasibility of SOA-based DSL implementation and its advantages over evolution/extension problems. Instead of utilizing the tools or artifacts from a traditional compiler context (e.g., lexical analysis, parsing, semantic analysis, and code generation [1]), our SOA approach uses WSDL to provide lexical analysis, describe syntactical and semantic specifications, and assist with syntax analysis. WS-BPEL has the capability to describe the programming and logical issues that may emerge in a DSL program. Furthermore, Web services can provide the implementation and semantic definition of domain-specific concepts.

Our six case studies suggest that SOA-based DSLs offer three implementation advantages:

- SOA, a loosely coupled software paradigm, addresses the extension and evolution problems of DSL implementation at the lexical, syntactic and semantic levels. For new, existing, extended, or evolved DSL constructs, lexical evolution can be achieved by automatically generated WSDL files, and semantic evolution can be achieved by introduction and/or composition of web services. Evolving or extending grammar productions (i.e., syntactic evolution) can be realized by reordering web service invocations;
- SOA offers interoperable communication among Web services implemented in different languages (e.g., Java and C# in our experiments), which addresses interoperability concerns. Different DSL language constructs may be implemented by different base languages and communicate with each other locally or remotely; and
- WS-BPEL, a language that our SOA-based DSLs utilize for programming, is a technology-neutral language that has been incorporated into the Oracle BPM, Eclipse, NetBeans, and Microsoft Visual Studio. Support from these vendors may reduce the effort to introduce tools for new or existing SOA-based DSLs. For example, debugging and testing tools [45] for specific DSLs may be easier
to implement using our approach because of the available debugging and testing support from these vendors. Additionally, two design and implementation burdens are eased using a SOA-based approach: (i) SOA offers improved modularization at the lexical, syntactical and semantic levels; and (ii) Lexical and syntax analyses, which traditionally are based on an interpreter or compiler-based DSL implementation, are delegated to the underlying WS-BPEL engine and no longer need to be handled explicitly for each new DSL.

This paper is organized as follows: Section 2 reviews the traditional approaches that support DSL implementation; Section 3 introduces the six DSLs implemented by our SOA-based approach; Section 4 discusses the implementation differences between the interpreter-based and SOA-based approaches, and summarizes the tradeoffs of the SOA-based approach; and concluding remarks are offered in Section 5.

2. Related work

To the best of our knowledge, there has not been any previous literature reporting on DSL implementation using a pure SOA-based approach (we note that Iwai et al. [29] considered the benefits of using WS-BPEL to represent models in an automotive domain context). Section 2.1 first reviews common DSL implementation patterns. Tools and frameworks that represent the traditionally accepted state-of-the-art approaches for DSL implementation, and frameworks that specifically target solving DSL evolution problems, are summarized in Section 2.2.

2.1. Methodologies for DSL implementation

This subsection reviews several DSL implementation methodologies suggested by Mernik et al. [37]. Six implementation patterns are introduced that may guide DSL developers in implementing a DSL.

(a) Interpreter/compiler patterns utilize the traditional compiler/interpreter techniques. Note that a compiler/interpreter need not be implemented from scratch, but can be implemented using the well-established compiler generators (e.g., CUP [19], ANTLR [41], LISA [38]). Because most DSLs are implemented using interpreter/compiler patterns (i.e., 42.1% [37]), this paper mainly compares the evolution/extension problems of SOA-based DSLs with those implemented using interpreter/compiler patterns;

(b) Embedding patterns introduce new DSL constructs into an existing host General-Purpose Language (GPL), such as Haskell, Java or C++;

(c) Among all DSLs, 13.2% [37] of them applied preprocessor patterns to translate DSL constructs into a base language. Because the Feature Description Language (FDL) [9] in Section 3.2.2 is implemented using such patterns, this paper also provides the evolution/extension comparisons between SOA-based and preprocessor-based FDLs;

(d) Extensible compiler/interpreter patterns add DSL optimization rules and code generation in the existing compiler/interpreter of a GPL. If such compilers/interpreters are implemented using a Metaobject Protocol (MOP) [22], extensions to semantics are possible;

(e) Commercial off-the-shelf (COTS) patterns utilize existing tools and/or notations for a specific domain. For example, to efficiently reuse existing CASE tools for program slicing, ACML (ANSI C Markup Language) [16] is capable of describing a useful, flexible and uniform data exchange format among the tools. OWL-Lite [36] is an example that borrows the notation and logic from
RDF (Resource Description Framework) and description logic, respectively. If web services are considered as COTS, the SOA-based approach presented in this paper is an example of this category; and
(f) A hybrid pattern is the combination of all of the above categories.

Regardless of which pattern is used, as long as there is tokenization, parsing, and semantics, the task of DSL evolution will be challenging.

2.2. Tools and frameworks for DSL implementation

This subsection highlights four notable tools for both domain-specific programming and modeling language implementation. Newly introduced frameworks that address the specific challenges of DSL evolution problems are also presented.

ANTLR (Another Tool for Language Recognition) [41] and CUP (Construction of Useful Parsers) [19] are compiler generators that generate a DSL’s parser, compiler, or interpreter based on the syntax and semantics of a DSL. Although the objectives and functionalities of ANTLR and CUP are the same, their internal mechanisms are quite different. Firstly, ANTLR contains the lexer generator as a built-in component, while CUP consumes tokens obtained from external lexical analyzers (e.g., lexers generated by JFlex [23]). Secondly, ANTLR utilizes top-down parsing to traverse DSL programs, but CUP applies bottom-up parsing instead. DSLs implemented by ANTLR and CUP require very different strategies to address DSL implementation and evolution due to such differences, which will be explained in more detail using case studies in Section 3.

LISA (Language Implementation System based on Attribute grammars) [38] is also a compiler generator. However, the design is very different from ANTLR and CUP. One of the design goals for LISA was to facilitate incremental language development [39] where language extensions can be specified easily. Instead of modifying language specifications (e.g., lexical, syntax and semantics) from scratch, a language designer can extend or override existing lexical, syntax and semantic specifications. Using the concept of multiple attribute grammar inheritance, language specifications can be reused and extended easily. Although such features clearly distinguish LISA from other compiler tools like ANTLR and CUP, a language designer still needs to concentrate on lexical and syntactic parts of a DSL. In contrast, with the SOA-based approach a language designer solely focuses on services, which implement the semantics of a language construct. Most of the DSLs presented in this paper (e.g., Robot DSL, FDL [9], and Video Store Language (VSL) [14]) have been implemented also in LISA.

In order to solve challenging evolution problems commonly seen in the DSL community, several frameworks have been introduced in recent years. Geest et al. [15] introduced a framework that tackles versioning issues when software implemented by SOA evolves. Specific kinds of evolution addressed in [15] include renaming and restructuring classes as well as removing inheritance and recursion. Karaila [21] reviewed the evolution history of FBL (Function Block Language), a DSL for describing real-time control programs for distributed environments. The specific focus of the study was how FBL and its development environment answered the evolution requests classified by Lehman’s laws, which describe how large software systems tend to evolve [21]. The NEVERLANG framework [4] introduced the composition concept that integrates language slices as basic units for composition. The definitions of syntax and semantics (called modules in [4]) are comprised in each slice. The slices are merged by syntactically connecting the non-terminals of two slices together. Introducing new statements, adding functionality to existing statements, and replacing semantics of statements are three kinds of DSL evolution activities described in [15]. There are also many other DSL implementation methodologies,
tools, and frameworks available (e.g., Visual Studio DSL tools [7], MPS [20], xText [11], AMMA [28], and GME [17]) that offer similar capabilities to the approaches just described.

Our contribution in this paper does not aim at introducing a framework or tool for SOA-based DSL implementation. Rather, a primary objective of the paper is to show the feasibility of using existing tools (in our case Oracle BPM, MicroSoft Visual Studio, and NetBeans) to implement SOA-based DSLs so that the effort to reinvent common infrastructure is not necessary. Supplementary tools or frameworks that may assist SOA-based DSL implementation may be introduced later as the approach becomes more popular and mature.

3. SOA-based DSL implementation

This section validates the feasibility of using SOA to implement DSLs by introducing six case studies representing imperative, declarative and hybrid DSL categories. Each section also discusses the benefits observed from SOA-based DSLs over their counterparts implemented by the traditional interpreter/compiler approach. Note that the work presented in this paper is an extension of [31,32], where one imperative and one declarative DSL are introduced.

3.1. Imperative DSLs

Imperative languages, designed around the von Neumann architecture, are composed of a sequence of statements that describe computation and state updates using assignment, conditional, and iteration statements, among others [42]. This subsection introduces two imperative DSLs: one for controlling simple robot movements and the other for controlling optimization and convergence of evolutionary algorithms.

3.1.1. Robot DSL

The Robot DSL is a simplified imperative DSL that is used to control robot movements and update the location coordinates of a robot. The DSL has been implemented using LISA (as described in [39]) and ANTLR (as described in [45]). Both the LISA and ANTLR implementations follow the compiler/interpreter pattern. An XML-based Robot DSL was also introduced, which is implemented using pure XML-related techniques (e.g., XML schema and XSLT) to compute coordinates. Programming in such a DSL is simple: a sequence of left, right, up or down movements with distance can be specified to navigate the robot to its destination. An example program of the Robot DSL is shown in the following code snippet. Its grammar can be found in [45].

```plaintext
1 Initial Position (0, 0);
2 Left 5;
3 Up 4;
4 Right 3;
5 Down 2;
6 Print Position;
```

For the SOA-based Robot DSL, each domain-specific statement shown in the above code snippet will be implemented as a web service, acting as a movement command that sends a specific direction with associated distance to a Lego Mindstorms NXT robot via USB or Bluetooth communication. Then, a DSL program written in WS-BPEL will describe a sequence of robot movements by linking such
web services. After the underlying WS-BPEL engine is invoked, the DSL program will send all the commands to the robot that interprets the commands and performs the movements accordingly. The SOA-based DSL program (written in WS-BPEL) and video demonstration to request the NXT robot to follow a square are available at [30].

We also implemented an alternative SOA-based Robot DSL that used code generation and Java Reflection. The “CompileJavaSource” web service first generates the source code of the aforementioned web services. Java Reflection [13] is then utilized to compile, load, and invoke the necessary files, including those for USB and Bluetooth communications obtained from a third-party. Our experiments show that web services generated by Java Reflection can be invoked successfully, such that XML messages can be marshalled and unmarshalled. USB and Bluetooth communications also perform successfully with the commands sent to the robot. The video demonstrations are also available at [30].

As for the comparison between SOA-based and non-SOA-based DSL implementation, there are several similarities that can be identified between the SOA-based and XML-based implementations of the Robot DSL. For the XML-based Robot DSL, an XML schema or DTD defines the grammar of XML files representing the DSL syntax. Then, XSLT is used to compute the coordinates of a robot by navigating and counting the number of “left”, “right”, “up” and “down” XML tags within an XML file. Similarly, for the SOA-based Robot DSL implementation, an XML file also conforms to its XML schema. The main difference is that XML files act as messages passed from direction-related web services to a robot, and a coordinate is updated by each web service invocation. In summary, the aforementioned SOA-based Robot DSL implementations show that converting from a compiler/interpreter-based Robot DSL to a SOA-based Robot DSL is straightforward. Each robot movement statement is transformed into a corresponding web service. Additionally, using Java Reflection for web service code generation and reusing third-party communication source code (wrapped as web services) are also feasible in a SOA-based DSL implementation. A more complex version of the Robot DSL will be introduced later in Section 3.3 by adding control statements. The advantages of using SOA to address the challenges of extension/evolution, interoperability and tooling support will be introduced in Sections 3.1.2 and 3.2.

3.1.2. **PPCeA**

PPCeA (Programmable Parameter Control for Evolutionary Algorithms) [34] is an imperative DSL used in the domain of evolutionary algorithms for controlling domain-specific parameters to balance between optimization and convergence. This section first reviews the interpreter-based PPCeA and then introduces the SOA-based PPCeA.

3.1.2.1. **Interpreter-based PPCeA**

PPCeA was implemented initially by following a compiler/interpreter pattern. The initial implementation contained a lexical analyzer generator for Java, namely JFlex [23], which was used to define and recognize the terminal symbols of PPCeA. Also, an LALR [1] parser generator for Java, namely CUP [19], defined the grammar of PPCeA and its semantics. Figure 1 shows a PPCeA program that iterates through five optimization search processes. Each process initializes the population and searches its optimal solution using \textit{init} and \textit{callGA}, respectively. The mutation rate (\(p_m\)) is reduced every Epoch generations (\(\text{Epoch} = 1\) in Fig. 1) until the maximum generation number is reached (\(\text{Maxgen} = 10\)).

The main implementation obstacle of the interpreter-based PPCeA emerged from frequent changes to the DSL grammar and evolution of the language constructs. When domain-specific constructs are extended or evolved, the corresponding lexical, syntactical and semantic parts require a considerable amount of effort to make corresponding changes. For example, if a new DSL statement (called \textit{crossover})
is defined, a terminal for crossover as well as the syntax and semantics for crossover need to be introduced in JFlex and CUP, respectively. Similarly, if an existing *mutation* is evolved from one-flip (i.e., swapping one bit, representing a gene, in a binary string) to \( n \)-flips, DSL developers are not only responsible for the changes occurring at the lexical, syntactical, and/or semantic levels, but they are also responsible for harmonizing such changes among these three levels. For example, suppose a “*mutation*” statement is evolved from flipping 1 bit to \( n \) bits. Its domain-specific statement is evolved to “*mutateNFlip*(int \( n \))”, where \( n \) is the number of flips. Tokenization for “*mutateNFlip*”, parsing for “*mutateNFlip*(int)”, and the semantics of how \( n \)-flips mutation works need to be respectively defined by JFlex, CUP, and a class or a method. Any inconsistency among these implementations will result in errors. Even worse, if DSL grammars require reordering rather than just addition/revision (as seen from the previous two examples), the complexity will be even higher. For example, assume that the “*callGA*” statement in Fig. 1 is further decomposed into “*crossover*”, “*mutation*”, “*selection*” and “*evaluation*” statements. This decomposition requires a new grammar production to describe the non-terminals with the associated order of the four decomposed statements in CUP. Additionally, tokens written in JFlex and classes/methods for such statements are needed. Any statement or order evolution/extension related to the grammar production will affect its tokens, grammar, and semantics in different magnitudes. Also, suppose a new domain-specific requirement specifies that the “*init*” statement can be invoked only before the “*callGA*” statement. In such a case, productions and semantics related to these two statements need to be carefully revised.

In addition to the aforementioned necessary changes, conformance to the underlying supporting parser generator is required. For example, the Robot DSL and PPCea utilize ANTLR and CUP, respectively, as parser generators. However, the parsing mechanism is quite different: ANTLR utilizes an LL(*) parser [1] to traverse a DSL parse tree from top-down with arbitrary lookaheads. If conflicts occur, left-factoring, substitutions, and left-recursion removal will be required [1]. Conversely, CUP applies an LALR parser to traverse a DSL parse tree from bottom-up. Shift/reduce and reduce/reduce conflicts need to be resolved so that CUP can generate the target parser of a DSL properly. Such an endeavor is not required in the SOA-based DSL implementation advocated in this paper. More comparative details will be discussed in Section 4.

```plaintext
1  genetic
2  Round := 5;  // # experiments
3  Epoch := 1;
4  r := 0;     // counter for # experiments
5  while ( r < Round ) do
6     g := 0;     // counter for # generations
7     tmp := 1.00; // tmp variable
8     Maxgen := 10; // maximum # of generations
9     init;     // initialize population
10    while ( g < Maxgen ) do
11        pm := (1.0 / 1250.0) + (0.042 / tmp);
12        // mutation rate is adjusted every few gen.
13        callGA;   // invoke GA process
14        tmp := tmp * 2;
15        g := g + Epoch
16        p <- pm in updated every Epoch (1 in this
17        case) generations */
18    end
19  end
20  genetic
```

Fig. 1. A PPCea program that adaptively adjusts mutation rate to reach optimization.
Table 1
Web services for PPCea

<table>
<thead>
<tr>
<th>Web services</th>
<th>Function objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize a population, configure domain-specific parameters</td>
</tr>
<tr>
<td>Select</td>
<td>Select offspring out of a population following different strategies</td>
</tr>
<tr>
<td>Mutate</td>
<td>Mutate individual(s) out of a population</td>
</tr>
<tr>
<td>Crossover</td>
<td>Reproduce offspring out of a population using crossover</td>
</tr>
<tr>
<td>Evaluate</td>
<td>Evaluate the result of a fitness function</td>
</tr>
<tr>
<td>Update</td>
<td>Update the values of domain-specific parameters</td>
</tr>
<tr>
<td>Entropy</td>
<td>Compute the randomness of a population</td>
</tr>
</tbody>
</table>

Fig. 2. WSDL of “Mutate” web service.

3.1.2.2. SOA-based PPCea

For implementing SOA-based PPCea, the process starts with introducing new domain-specific web services that act as domain-specific statements. Based on our domain analysis, seven types of web services, shown in Table 1, have been introduced. Each web service is implemented using Java Enterprise Edition [44] with the corresponding generated WSDL file. Figure 2 shows the partial WSDL of the domain-specific web service for mutation (“Mutate”).

Some domain-specific parameters (e.g., Epoch, Maxgen, Round, and pm in Fig. 1) are used to control
the optimization process in order to balance the optimal result with the convergence rate. Other temporary parameters in PPAe (e.g., \( r \), \( g \), and \( t m p \) in Fig. 1) can be defined by an XML schema within the SOA-based PPAe program written in WS-BPEL. For example, \( p m \) is declared as an XML floating-point type that the BPEL engine recognizes. Non-primitive domain-specific parameters, such as \( p p a e \) in PPAe (i.e., a set of individuals, each of which is composed of a sequence of genes, represented by an integer or Boolean), may be declared using complex types supported by an XML schema. Figure 3 shows the XML schema of \( p p a e \). Because WS-BPEL has provided other structure activities (e.g., <if>, <repeatUntil> and <forEach> [40]) and is able to introduce domain-specific parameters, writing SOA-based PPAe programs in WS-BPEL is a viable solution.

Figure 4\(^1\) shows a SOA-based PPAe program written in WS-BPEL (i.e., a SOA-based PPAe program) as Fig. 1’s counterpart. Note that Fig. 1 (i.e., interpreter-based PPAe) does not have an “Eval” statement because it is part of “Init” and “callGA”. For better modularization, the SOA-based PPAe implementation introduces a stand-alone “Eval” web service. Figure 4 starts by performing <invoke> of “Init” and “Eval” web services. Then, a series of “Select”, “Crossover”, “Mutate”, “Eval”, “Entropy”, and “Update” web services are invoked.

The advantages of SOA to address DSL extension and evolution problems can be observed also in Figs 2 and 4: The “Mutate” web service, comprising a @WebMethod with \( p m \) and \( p p a e \) parameters, was originally designed to flip only one Boolean value of an individual. If this web service is to evolve to \( N \)-point mutation, flipping \( n \) Boolean values instead, the web service needs to: (a) Edit the semantics of \@WebMethod(operationName= "MutationService") to \( N \)-point mutation; and (b) Add (@WebParam(name= "nPPoint") int nPoint) as an annotation to its \@WebMethod. The new WSDL (Fig. 2) in compliance with this evolution will be updated automatically, and DSL users only

\(^1\)For resolution and size reasons, Fig. 4 was edited by collapsing web services and domain-specific parameters, as well as adding arrows. Please refer to [30] for the original figure.
need to pass a value to \textit{nPoint} at the WS-BPEL level. Namely, lexical and syntactic evolution is done automatically, and semantic evolution from 1-flip to \textit{n}-flip is isolated/modularized in the “mutate” web service. Similarly, if a new crossover is introduced (i.e., \textit{extension}), a crossover web service must be implemented accompanied by an automatically generated WSDL. End-users could easily invoke new mutation and crossover web services, acting as domain-specific statements, using WS-BPEL with partner links, each of which defines an interaction between a WS-BPEL process and the involved web service. Most importantly, if PPCea’s grammar is reordered, there is no influence on the WSDL or web services. The required change is a revision of the SOA-based PPCea program written in WS-BPEL so that web services can be executed according to the new grammar/order. For example, the two grammar production reordering examples shown in Section 3.1.2.1 can be done simply by reordering the invocation orders of web services in the SOA-based PPCea program. No other parts will be influenced.

Yet, if such changes/additions/reorder were made to an interpreter-based implementation of PPCea, changes and consistency at JFlex, CUP and Java as well as conformance to CUP would need to be considered. Although some parser generators (e.g., ANTLR [41]) may combine lexical analysis and syntactic analysis together to ease the burden that JFlex and CUP exhibit, our approach shows that the SOA approach will update the corresponding files in compliance with the extension or evolution at the lexeme, syntax or semantics levels. Namely, the SOA approach offers \textit{well-modularized} abstractions at the lexical, syntactical and semantic levels: Web Services define the semantics of DSL statements; WSDL describes the specifications of such services that act as the results of lexical analysis and assist
syntax analysis; and WS-BPEL depicts a DSL program where the underlying BPEL engine performs syntax analysis following a predefined yet flexible WS-BPEL grammar. Most importantly, as seen in the above examples, extension/evolution of SOA-based DSLs always starts from web services: If the extension/evolution is semantics-related, such changes/additions will be isolated in web services (as seen in the two n-flip mutation implementation alternatives and crossover). However, if tokens or syntax of domain-specific statements require extension/evolution, changes are still initiated from web services through annotation modifications (e.g., @webMethod and @webParam). The associated WSDL files will then be updated automatically without affecting other parts. If semantic changes are also required in addition to lexical or syntactical changes, they will continue to be isolated in the associated web services. If DSL grammars are reordered, as long as WS-BPEL's grammar can describe them, then the SOA-based DSL implementation does not need to be concerned about such evolution. As for domain-specific parameter passing among web services, extension/evolution will target changes of the XML schema and its marshalling and unmarshalling operations. Yet, one deficiency of the SOA-based DSL implementation emerges when extensions/evolutions are applied to WS-BPEL's existing tokens, syntax, and semantics, which will require customization of the underlying WS-BPEL engine. However, such a request rarely occurs due to WS-BPEL's flexibility and generality.

3.1.2.2.1 Alternative implementations

After showing the benefits of a SOA-based DSL implementation to support extension/evolution, the following paragraphs discuss how a SOA-based DSL implementation offers support for interoperability and tooling. Three additional implementation alternatives of Fig. 4 are specifically applied to marshalling and interoperability. Note that all the web services are run under the same machine. Hence, execution time that may be affected by network load is omitted in the following experiments.

1. The first alternative utilizes the Java Architecture for XML Binding (JAXB) [43] to marshal and unmarshal messages passed among web services. JAXB binds the XML schema when performing marshalling and unmarshalling. Additionally, symbol tables used to store DSL parameters are public to all web services. Note that although the number of parameters passed among web services may be reduced, such an implementation violates SOA ideology.

2. The second alternative replaces JAXB by the Streaming API for XML (StAX) [6] to marshal and unmarshal messages. The symbol table approach is also applied. The main difference between Experiment (2) from Experiment (1) in Table 2 is that StAX adopts a “pulling” approach that offers more parsing control and efficient processing [6].

3. The third alternative implements interoperability between Java-based and C#-based web services – all but the “Init” web service is implemented in Java. These web services are partner-linked by WS-BPEL under the Oracle BPM. Additionally, to advocate SOA ideology, symbol tables are replaced by XML message passing. Namely, user-defined parameters are stored in XML and validated by the XML schema. Similar to Experiment (1), this alternative utilizes JAXB to validate and exchange XML messages. The left box of Fig. 5 shows a partial snapshot of Oracle BPM to execute SOA-based PPCea. The right box of Fig. 5 is a partial C# code snippet of the “Init” web service invoked by the left box. The demonstration videos and source code are available at [30].

Table 2 summarizes the performance and average experimental results of Fig. 1 (row 2) and three implementation alternatives of Fig. 4 (rows 3 to 5) under the same parameter and configuration settings. As shown in columns 4 to 6 (i.e., Best Fit, Avg. Fit, and Worst Fit), all experiments have relatively close optimization search results. Such results suggest that Figs 1 and 4 are executed in the same manner even though the implementation paradigm is changed from interpreter-based to SOA-based. As for columns
Table 2

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Exec. time (sec)</th>
<th>Memory usage (Byte)</th>
<th>Best fit</th>
<th>Avg. fit</th>
<th>Worst fit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>5.00</td>
<td>7,612,457</td>
<td>19.39</td>
<td>19.53</td>
<td>19.59</td>
</tr>
<tr>
<td>(1)</td>
<td>12.00</td>
<td>9,259,575</td>
<td>20.61</td>
<td>20.86</td>
<td>21.03</td>
</tr>
<tr>
<td>(2)</td>
<td>9.40</td>
<td>10,007,744</td>
<td>20.64</td>
<td>20.86</td>
<td>21.04</td>
</tr>
<tr>
<td>(3)</td>
<td>53.82</td>
<td>2,058,264,576</td>
<td>20.43</td>
<td>20.78</td>
<td>20.96</td>
</tr>
</tbody>
</table>

2 and 3, tradeoffs of SOA-based PPCea can be observed easily – both execution time and memory usage are worse using a SOA approach compared to interpreter-based PPCea. More details of such tradeoffs will be discussed and analyzed in Section 4.

3.2. Declarative DSLs

Different from imperative languages that utilize DSL control constructs to determine how a DSL program is executed, domain-specific declarative languages instead describe what a program should do when executed by specifying the relationships between DSL statements. This section introduces two declarative DSLs: One for computing the rental charges of a video store and the other for computing different configurations out of a set of features. Because of the semantics rules defined for them, changing the statement orders of the two declarative DSLs will not affect the execution results, which can be also seen in many other declarative languages.

3.2.1. Video store language (VSL)

The Video Store Language (VSL) is a declarative DSL that has been implemented using Java in [10] and LISA in [25], both following the compiler/interpreter pattern [37]. The language is used to compute

```csharp
namespace PPCeaWebService
{
    using System;
    using System.Collections;
    using System.ComponentModel;
    using System.Data;
    using System.Linq;
    using System.Net;
    using System.Web;
    using System.Web.Services;
    using System.Xml.Linq;

    {
        [WebMethod]
        public string SelectInitialization(int initType, InitFeature config)
        {
            switch (initType)
            {
                case 1:
                    return new InitDefault().InitFeature(config);
                default:
                    //just return the config as it is
                    return config;
            }
        }
    }
}
```
the total charges/income of the movies rented by the customers of a video store. Figure 6 shows the grammar of VSL adapted from [25] and a simple program example conforming to the VSL grammar. The example program expresses that the video store currently has three movies in stock: 5 copies of “Star Trek”, 10 copies of “Inception”, and 3 copies of “Up.” The classifications of “Regular”, “new”, and “child” represent three price categories for movies. Customers “Joseph”, “Charles”, and “Mary”, currently have 15.00, 20.00, and 25.00 dollars remaining in their accounts, respectively. The DSL program states that “Joseph” rented “Inception” for three days and “Mary” rented “Up” for two days. After program execution, each customer’s remaining amount, quantity of movies, and the total charge/income of the store are updated.

To implement the SOA-based VSL, our first step is to introduce the example program shown in Fig. 6(b) in XML, acting as messages/data transferred between web services. Figure 7 shows the XML schema of the example program that conforms to the grammar listed in Fig. 6(a). The next step is to introduce four web services: “MovieWS” acts as the movie production, listed in line 2 of Fig. 6(a), which consumes the XML messages of movie statements in Fig. 6(b) (e.g., lines 1 to 3). As a movie-related XML message is received, “MovieWS” will append movie-related XML messages until no more movies are added. Similarly, “CustomerWS” and “RentalWS” are web services that sequentially append customer-related messages (lines 4 to 6 of Fig. 6(b)) and rental-related messages (lines 7 and 8 of Fig. 6(b)), respectively. Lastly, quantities of movies, the remaining amount of customers, and total income of the video store should be updated accordingly. For simplicity, such computation is introduced as the “totalIncome” web service that generates an XML output containing the values of the aforementioned variables.

Figure 8 shows the WS-BPEL source code that acts as the counterpart of the VSL example program in Fig. 6(b). As seen in the figure, three movies are first added by “MovieWS” and then customers and rentals are added by “CustomerWS” and “RentalWS”, respectively. Lastly, “totalIncome”, acting as a compilation of previous DSL statements, is invoked to update necessary variable values. Please visit [30] for the source code, snapshots, XML input and output files, and a video demo of VSL. Note that Fig. 8 does not mean that a computational result can be obtained only at the end of the program after “totalIncome” is invoked. To obtain the computational results after “MovieWS”, “RentalWS”, or

| 1 VIDEO_STORE -> MOVIES CUSTOMERS RENTALS |
| 2 MOVIES -> MOVIES MOVIE |
| 3 MOVIE -> title PRICE QUANTITY |
| 4 CUSTOMERS -> CUSTOMERS CUSTOMER |
| 5 CUSTOMER -> name INCOME |
| 6 RENTALS -> RENTALS RENTAL |
| 7 |
| 8 RENTAL -> daysRented MOVIE |
| 9 PRICE -> new | child | reg |
| 10 QUANTITY -> integer |
| 11 INCOME -> double |

Fig. 6. The grammar (top) and example program (bottom) of VSL adapted from [25].
“CustomerWS” is invoked. “totalIncome” can be invoked right after any one of the three web services. Namely, “totalIncome” may also be composed with the three other web services if needed. Our primary reason to implement “totalIncome” as a standalone atomic web service is for modularization and flexibility purposes. In either case (Fig. 8, invocation after each web service, or composed web services), its language characteristics (i.e., what a program does and no influence on DSL statement orders) are all retained.

The solutions to address VSL evolution and extension are the same as those introduced in the Robot DSL and PPCea examples discussed in Section 3.1.2.2: Semantic evolutions/extensions are isolated in web services; syntactical and lexical evolutions/extensions are independently achieved by annotations of web services and WSDL; and domain-specific parameter evolutions/extensions purely focus on XML schema and corresponding marshalling and unmarshalling operations. As for DSL grammar evolution
and extension (i.e., reorder), the following example shows the advantage of using SOA to implement declarative DSLs. Suppose that the first production of the VSL grammar in Fig. 6(a) is reordered from

\[
\text{VIDEO\_STORE} \rightarrow \text{MOVIES CUSTOMERS RENTALS}
\]

to

\[
\text{VIDEO\_STORE} \rightarrow \text{RENTALS MOVIES CUSTOMERS}
\]

and the remaining productions are unchanged. Because such a reorder does not affect the underlying
WS-BPEL, the only change needed is to reshuffle the web services so that “RentalWS” will be invoked first, followed by “MovieWS” and then “CustomerWS”. Our justifications are as follows:

1. As mentioned before, a declarative DSL describes what a program should do and the results depend on the relationships between DSL statements. The purpose of VSL is to perform inventory management of movies, customers, and rentals; namely, the existence and quantities of these three types of objects. The (re)order of the objects to be appended by “RentalWS”, “MovieWS”, and “CustomerWS” will not affect the final output generated by “totalIncome”.

2. Similarly, if we introduce a new DSL statement, called returnWS, which acts as returning movies from customers, the quantities of such movies will be deducted accordingly. The order of web service invocations still does not matter.

In summary, for SOA-based VSL, production reorder related to non-terminals will result in the reorder of web service invocations. The output is unchanged because the order of invocations is not about how a DSL program is executed, which also advocates the characteristics of some declarative languages. However, for production reorder related to terminals (lines 3, 5 and 8), changes are isolated in the XML schema and related marshalling/unmarshalling, because such productions with terminals define domain-specific parameters, which are converted to XML messages in SOA-based VSL (i.e., the first step mentioned earlier). If a DSL grammar does not specify the order of different kinds of domain-specific parameters, as seen in Robot DSL, PPCea, and FDL described next, the effort to evolve XML schema and the associated marshalling/unmarshalling is not needed. For interpreter-based DSLs, such evolution/extension on domain-specific parameters requires a great amount of work. Discussions and solutions on how to tackle such evolution/extension can be found at [33]. Lastly, both kinds of production reorder do not require any conflict resolution as seen in the SOA-based Robot DSL and PPCea examples.

3.2.2. Feature description language (FDL)

The Feature Description Language (FDL) [9] is a declarative language that textually describes feature diagrams for domain analysis. The language introduces all-of, one-of, more-of and optional feature operations to explore all possible configurations along with requires, excludes, include and exclude constraints to reduce the number of possibilities.

3.2.2.1. Preprocessor-based FDL

In [24], a preprocessor pattern-based FDL is introduced that translates its DSL constructs into Java code. The Java code consists of a list of user-defined feature and constraint classes. The Java interpreter performs semantic analysis and invokes a predefined rule manager class, comprising a set of normalization, variability, expansion and satisfaction rules [9] to generate the resulting configurations. For example, Fig. 9 shows a preprocessor-pattern-based FDL program that describes different configurations of a car. After executing such a program, a number of combinations of a car can be derived.

3.2.2.2. SOA-based FDL

To implement SOA-based FDL, the first step is to introduce XML messages that will be consumed by web services. Figure 10 shows an XML message manually converted from line 2 of Fig. 9. After this step, two web services are introduced: (a) “FeatureWS” consists of eight operations, namely SOA-based
FDL statements, (i.e., all-of, one-of, more-of, optional, requires, excludes, include and exclude). Each operation will invoke corresponding internal rules [9] along with two XML messages – one is a new input as in Fig. 10 and the other is the combinatory XML result generated from the previous operation (if any). A new set of configurations in XML will be generated, which may be taken as the input of a subsequent operation. Lastly, “CompileWS” prints out all possible alternatives. Similar to “totalIncome” of VSL, “CompileWS” can be implemented as an atomic or a composed web service. It can be invoked anywhere to obtain the results.

Writing a SOA-based counterpart of Fig. 9 is relatively easy. What is required is to partner-link “FeatureWS” operations and introduce XML messages for each operation to consume (which will be automated in the future). Figure 11 shows a partial WS-BPEL snapshot (“FeatureWS” only) that will generate six constrained possibilities and be printed by “CompileWS”. Note that similar to all the DSLs introduced earlier, there is no need to implement tokenization and parsing processes for SOA-based FDL. More details will be discussed in Section 4.

To illustrate the advantage of the SOA approach to address the DSL evolution problem, two FDL operations are introduced. The first one is the none-of operation, which identically acts as an exclude constraint. Our intention for introducing this operation is to show that if a syntactic (but not semantic) evolution of a SOA-based DSL is triggered, the SOA-based FDL can evolve by utilizing a newly generated WSDL without affecting other parts of the implementation. As for syntactic and semantic evolution, the two-of operation is prototyped. For this operation, syntactic evolution is achieved by automatically generated WSDL. As for semantic evolution, most of the rules for the one-of operation are reused. A new rule that can be realized as a web service is introduced to generate a set of two combinatory atomic features out of a feature list. Hence, semantic evolution may be considered as introduction and/or composition of web services, which reflects SOA’s advantages. As for evolution on grammar reorder, similar to the SOA-based VSL, the SOA-based FDL will not be affected as long as the changes
do not affect the grammar of the underlying WS-BPEL engine. All SOA-based FDL experiments and demonstrations are available at [30].

3.3. Hybrid DSLs

DSLs that are constructed by mixing original or existing DSLs with GPLs are classified into the hybrid\(^3\) DSL category [3] [45]. Within such a category, a hybrid DSL can be further identified by either embedding a DSL into a host GPL (e.g., Swing User Interface Language (SWUL) [3]), or conversely, embedding a GPL into a host DSL (e.g., a Hybrid Robot language [45] that embeds Robot DSL with Java Swing to display the coordinates of a robot). A primary objective of hybrid DSLs is to advocate reuse from existing DSLs/GPLs so that the endeavor to introduce new features to a DSL can be reduced. The following subsection introduces a SOA-based hybrid DSL, called HYROL (HYbrid Robot dsL), which controls movements of a robot by mixing WS-BPEL with Java. Note that because the main functionality

\(^3\)Note that a hybrid DSL is different from the hybrid pattern introduced in [37]. The former one is from the perspective of embedding DSLs and GPLs, and the latter one is from the perspective of mixing different implementation patterns when a DSL is implemented.
and objective of HYROL are different from the hybrid Robot language introduced in [45], a new name is given for identification purposes.

3.3.1. HYROL

To demonstrate the feasibility of the SOA approach applied to hybrid DSLs, this subsection introduces HYbrid Robot dSL (HYROL). As a variation of the Robot DSL described in Section 3.1.1, both Robot DSL and HYROL share the same objective: to control movements of a robot given distances and directions. The main difference between the two DSLs is that the Robot DSL is purely written in WS-BPEL, while HYROL (both versions) is written in a way that mixes WS-BPEL with Java. Our first version of HYROL (i.e., HYROL1) can be programmed in the following way: WS-BPEL is used as a host language to describe web service invocations, and Java/leJOS (a subset of Java for programming Lego Mindstorms NXT brick) [35] as a mixed language to describe programming logic. Because WS-BPEL is the host language, HYROL1 needs to be converted into pure WS-BPEL before execution. The Java/leJOS control statements must be converted into equivalent WS-BPEL control constructs expressed in XML (e.g., <if> and <while>) without changing the DSL statements expressed in WS-BPEL (e.g., <invoke partnerLink >) (HYROL1’s source code is available at [30]). The underlying WS-BPEL engine then executes the converted WS-BPEL program. The advantages of extension/evolution that other SOA-based case studies possess are retained because of utilizing the WS-BPEL engine.

The second version of HYROL (i.e., HYROL2) employs Java/leJOS movement and sound statements, and embeds WS-BPEL control statements (e.g., if-else, while) to determine the programming logic. HYROL2 describes movements of a robot by utilizing Java/leJOS as the host language and blending it with WS-BPEL. The following paragraphs describe how HYROL2 is implemented in detail.

The question before undertaking any concrete implementation is, “Who is responsible for executing HYROL2?” All of the previous case studies utilized a WS-BPEL engine to perform DSL program execution. However, because leJOS statements are embedded into HYROL2, the underlying WS-BPEL engine can neither recognize nor execute such statements. To solve such a problem, there is a need to convert HYROL2 programs into a uniform format that can be executed by a target engine/interpreter/compiler (suggested by the preprocessor pattern [37]). Because HYROL2 is a Java-hosted DSL, a Java interpreter provides a solution to this challenge, as noted by the following four steps: (1) Introduce a HYROL2 program written in both WS-BPEL and Java; (2) Transform the HYROL2 program into a Java-like XML program (described later); (3) Parse the Java-like XML program; and (4) Interpret and execute the Java-like program. Note that converting HYROL2 to pure Java instead of Java-like XML is a feasible and simple solution. Our rationale for implementing a Java-like XML solution is to be consistent with WS-BPEL’s XML format and increase the level of difficulty of the project.

A HYROL2 program that blends WS-BPEL with Java is shown in Fig. 12. The program requests a robot to keep moving east if the moving distance is smaller than a predefined variable, maxDistance. Otherwise, an alarm embedded in the Lego Mindstorms NXT brick will sound. Note that all but direction- and sound-related statements are written in WS-BPEL (i.e., the statements using <xs:string>). As for Java/leJOS statements, they conform to Java syntax and are embedded between WS-BPEL constructs.

The next step is to perform program transformation. Figure 13 shows part of the XML schema of the target files after transformation. As seen in the figure, Java-like structures are introduced. For example, the “class” complex type, consisting of class name and “Parameters” and “Methods” complex types, acts as a counterpart of a Java class; “Parameters” comprises a collection of “Parameter” that consists of the identifier, data type, and value needed in a Java parameter; “Methods” comprises a collection of “Method” that consists of the information of the class it belongs to, method name, arguments and “logic”
it belongs to if the “Method” is embraced by a condition statement. Each “logic” is also associated with complex types. For example, “ifElse” logic consists of an operator (e.g., > or <), left and right operands, and methods associated with then and else parts, respectively. Currently, the XML schema does not allow expression of recursive and nested statements. This will be our future work. Figure 14 shows an example of a Java-like program that conforms to Fig. 13.

After a Java-like XML program is introduced, a StAX-based parser is needed to identify and parse XML elements of all the programming constructs (e.g., classes, methods, parameters and logic). Within the parser, each parsed XML element will be unmarshalled into Class objects [13,22], which will be used in the final step. Finally, the interpreter utilizes Java Reflection to perform Java/leJOS object initializations and invocations. The source code of HYROL2 and its interpreter, snapshots, and video demonstrations are available at [30].

Lastly, although the above HYROL2 program assumes that all direction- and sound-related statements are Java-based, it does not mean the hybrid approach cannot take care of those with only partial hybrid statements. For example, suppose “east” and “west” are Java statements that can be embedded into HYROL2 programs. To make the programs executable, we need to convert WS-BPEL of “north”, “south”, and “sound” into Java-like XML format. The web services associated with “north”, “south”, and “sound” statements need to be converted to the host language, too.

Because HYROL2, using Java as a host language, does not totally conform to the SOA-based approach, the roles that WSDL, WS-BPEL, and web services play at the different levels may not entirely hold. First, lexical analysis is still not needed, because Java-like XML files are transformed from original hybrid WS-BPEL files – all but Java-related lexemes are already tokenized before program transformation. Java-related lexemes are processed as a manual transformation. Syntactical analysis is performed by the verification of Java-like XML programs against the XML schema shown in Fig. 13. If any programming construct does not follow the schema, the programs will not be parsed by the StAX parser and no Class objects will be generated for the later Java Reflection-based execution. As for semantics, Java objects are instantiated and invoked by Java Reflection, which is different from the SOA-based approach that uses web services for semantic purposes. Regarding extension and evolution, lexical and syntactical additions/revisions and grammar reorder is isolated in the StAX parser and the XML schema. For example, to introduce nested if-else statements, a “logic” complex type needs to be revised. Lastly, semantic changes will be isolated in Java/leJOS code. Consistencies among schema, parser, and Java code is requested, which is similar to DSLs implemented using interpreter or compiler patterns.
An additional question may be asked regarding the ability of the hybrid approach to address the needs of a declarative DSL implementation. To understand this capability, reconsider the FDL as an example. Suppose one-of is no longer a WS-BPEL recognized statement, but instead it becomes a Java statement that can be mixed into a SOA-based FDL program. If WS-BPEL is chosen as the host language, a Java statement has to be converted back to the WS-BPEL-compiled statement and the one-of operation needs to become a web service. Conversely, if Java is the host language, the transformation will be similar to HYROL2. After the transformation, for both examples, a set of normalization, variability, expansion and satisfaction rules will be executed as in Section 3.2.2. Hence, the result will be the same. Lastly, an observation worth mentioning is that the Java-like XML schema and Java Reflection approach also provide an ability to describe and execute the hybrid Robot DSL [45], as well as other hybrid examples like SWUL [3], due to the equivalent structures between XML schema and Java syntax.
4. Discussions and lessons learned

This section discusses the commonalities and variabilities between the compiler/interpreter-based and SOA-based DSL implementations. The discussions are categorized based on compiler perspectives [1].

4.1. Lexical analysis and symbol tables

In this phase, an interpreter-based DSL utilizes a lexical analyzer to tokenize a DSL program into lexemes. Such analyzers include those generated by JFlex and related tools. Conversely, a number of XML tags and XML schema used for SOA-based DSL implementation have been defined in the WS-BPEL specification [40] and supported by the underlying WS-BPEL engine. For all of the DSLs introduced in this paper, there is no need to utilize a lexical analyzer. Additionally, because WSDL is used to describe the specifications of a web service, it may also be regarded as a tokenized string that represents a domain-specific statement used to invoke corresponding web service(s). For example, when a SOA-based DSL needs to invoke a domain-specific statement, WS-BPEL must provide a hook to the corresponding WSDL using a partner link.

A symbol table is a containment data structure for a compiler to “keep track of scope and binding information about names” [1]. For interpreter-based DSLs, a symbol table is usually internally implemented as a hash table comprising the aforementioned information. For SOA-based DSLs, because each
DSL construct may not be deployed to the same environment, a commonly shared internal hash table is not feasible. SOA-based PPCea’s Experiments (1) and (2) mimicked the interpreter-based symbol table implementation by introducing a publicly global hash table accessed by all web services. However, because of the deployment concern, such an approach may not be feasible in all cases. More importantly, SOA ideology does not encourage this implementation style. Experiment (3) instead passes the XML messages that contain symbol table information among web services. Then, each web service creates an internal hash table to interpret, validate (against an XML schema, if available) and store necessary variable information. All other SOA-based DSLs do not introduce symbol tables due to their simplicity.

4.2. Syntactical analysis

Syntactical analysis examines a DSL program and constructs its grammatical structure based on the DSL grammar. ANTLR, CUP and many other parser generators have been applied to construct compiler- and interpreter-based DSLs (e.g., ANTLR for the Robot language and CUP for PPCea). The main effort for DSL developers is to specify compiler/interpreter-based DSL grammars in compliance with the selected parser generators. For example, ANTLR is a top-down parser generator, but CUP is a bottom-up parser generator. As for our SOA-based DSL implementation, because all DSL programs (except for HYROL2) conform to the underlying WS-BPEL engine (i.e., they follow WS-BPEL grammar), endeavors to reinvent SOA-based DSL grammars and parsers are not needed. WSDL also plays an important role in syntactical analysis: because specification of a web service/domain-specific statement can be found in WSDL, “manual parsing” that determines if the workflow or a specific web service is reasonable and meaningful in a WS-BPEL may be performed by SOA-based DSL implementers. However, such manual parsing also exposes a critical pitfall: if a SOA-based DSL user is not knowledgeable, potential semantic errors may occur in a WS-BPEL program due to incorrect usage or improper ordering of web service invocations violating potential dependency injection (which may result from the inability to enrich the WS-BPEL grammar). For example, a SOA-based PPCea user may forget an “Eval” service in the workflow so that no fitness results can be obtained. Such errors may be avoided if grammars are richly defined. HYROL2’s syntactical analysis is totally different from other SOA-based DSLs. The details were covered in earlier parts of this paper.

4.3. Semantics and type checking

As mentioned in Section 1, Kosar et al. [24] and Mernik et al. [37] have analyzed several implementation patterns. Such patterns are mainly categorized based on how DSL semantics are realized. For example, the interpreter-based PPCea defines DSL semantic constructs, including control flows, as Java classes, which will be instantiated and then jointly function as an interpreter to execute PPCea programs.

The SOA-based Robot language is a very simple example to illustrate that a web service allows sending distance and direction commands to a Lego robot. As for the SOA-based PPCea, VSL, FDL and WS-BPEL-hosted HYROL1, domain-specific statements are wrapped as one or more web services. Implementation of such web services is not much different except that an internal commonly shared symbol table is no longer valid. Each web service now owns its symbol table, whose information is passed from other web service(s). Investigation on analyzing the scope of domain-specific parameters is needed – only those parameters that will be needed by most web services will be encapsulated in an XML message. There also is a need to introduce efficient marshalling and unmarshalling algorithms to parse the aforementioned XML messages. JAXB is a more formal approach: an XML schema is used to validate and convert between objects and XML instances. Conversely, StAX is a more casual
but efficient approach that processes XML as a stream and ignores tree construction. As for HYROL2, because Java is its host language, the focus on semantics and type checking would be on the interpreter using Java Reflection, as described in Section 3.3.1.

Several important advantages of SOA-based DSLs regarding semantics have been illustrated earlier (e.g., extension and evolution). Section 3.1.1.2 demonstrated an implementation alternative against the interoperability issue presented in [18]. Additionally, using the SOA-based approach, DSLs now can be implemented by more than one deployment environment. Because WS-BPEL is technology-neutral, SOA-based DSLs written in WS-BPEL can be realized in different IDEs such as Eclipse, .NET, NetBeans, and Oracle BPEL Process Manager.

4.4. Tradeoffs

Although the SOA-based approach offers solutions to extension, evolution, interoperability, and tool support through vendors, there are several major tradeoffs that require further improvement:

1. Usability: The interpreter-based PPCea program is easy to edit and perform experimentation. Although the WS-BPEL PPCea appears simple, it takes considerable time to construct. Declaring domain-specific parameters at the WS-BPEL level requires XML schema knowledge, and the situation becomes burdensome as the size of the specification increases. For example, numerous assignment links for assigning values from WS-BPEL to selected Web services need to be introduced. Some researchers are working on converting WS-BPEL to a programmer-intuitive scripting language, and vice versa, to ease such a burden [2].

2. Usage of Time and Resources: As shown in Table 2, experimental results of SOA-based PPCea take more time and memory usage than the interpreter-based approaches. One of the primary reasons is that marshalling/unmarshalling of requests represent bottlenecks, which are unfortunately mandatory components to realize Web services and offer interoperability. StAX is currently one of the fastest XML parsing techniques, yet may require further improvement to accommodate large volumes of data.

3. Over-Exposure: Because there is no symbol table, domain-specific parameters in XML messages are exposed to end-users. The benefit is that users have direct access to its domain-specific parameters and the right to change these parameters if needed. Yet, the continuing challenge is that misuse or potential pitfalls may be introduced by end-users if XML schema validation is not involved.

4. Over-Flexibility: Although the flexibility and generality of the WS-BPEL grammar facilitate SOA-based DSL extensions and evolutions with ease, they also result in potential pitfalls that DSL programs may be described incorrectly by DSL users. For example, some language constructs may require dependency injection, and others may be removed when the DSL grammar evolves. Constraints applied to WS-BPEL grammars are needed to reduce the chances that DSL programs are wrongly specified.

5. Conclusion

This paper introduces a new DSL implementation approach using SOA techniques: WSDL is used for lexical analysis and specifying domain-specific statements; WS-BPEL can be used to depict DSL programs against predefined WS-BPEL/domain-specific grammars; Web services are responsible for
describing semantics of domain-specific statements; and the XML schema defines and validates domain-specific parameters. The loosely coupled and technology-neutral features of SOA not only address the DSL extension/evolution problem, but also offer interoperability and potential tool support among different vendors. Improved modularization and delegation of the need for tokenization and parsing to the underlying WS-BPEL engine are additional advantages. Yet, tradeoffs surrounding WS-BPEL usability, bottlenecks on XML parsing time, and exposed domain-specific parameters still require additional work. Additionally, because of the flexibility of WS-BPEL grammars, the SOA-based DSL implementation approach assumes that DSL extensions/evolutions will not affect the underlying WS-BPEL’s lexemes, syntax and semantics based on our experiences in the six case studies. Introducing mechanisms to customize the underlying WS-BPEL engine with ease when extensions/evolutions occur is also an important future task. With this paper, we hope to raise more research interest in this direction so that novel techniques will overcome the tradeoffs and shortcomings of the current SOA-based DSL implementation.

References

S.-H. Liu et al. / Introducing domain-specific language implementation using web service-oriented technologies


