NeoIDL: A Domain-Specific Language for Specifying REST Services

Rodrigo Bonifácio*, Thiago Mael de Castro†, Ricardo Fernandes†, Alisson Palmeira†, and Uirá Kulesza‡
* Departamento de Ciência da Computação, Universidade de Brasília, Brazil
†Centro de Desenvolvimento de Sistemas, Exército Brasileiro, Brazil
‡Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte, Brazil

***Abstract*—Service-oriented computing has emerged as an effective approach for integrating business (and systems) that might spread throughout different organizations. A service is a unit of logic modularization that hides implementation details using well-defined contracts. However, existing languages for contract specification in this domain present several limitations. For instance, both WSDL and Swagger use language-independent data formats (XML and JSON) that are not suitable for specifying contracts and often lead to heavyweight specifications. Interface description languages, such as CORBA IDL and Apache Thrift, solve this issue by providing specific languages for contract specifications. Nevertheless, these languages do not target to the REST architectural style and lack support for language extensibility. In this paper we present the design and implementation of NeoIDL, an extensible domain specific language and program generator for writing REST based contracts that are further translated into service's implementations. We also describe an evaluation that suggests the rapid return on investment with respect to the design and development of NeoIDL[1].**

I. Introduction

Service-oriented computing (SOC) [6] is a consolidated approach that enables the development of low coupling systems, which are able to communicate to each other even across different domains. Thanks to the use of open standards and protocols (such as HTTP and HTTPS) in SOC, service orchestration enables the automation of business processes among different corporations. A service is defined as a unit of logic modularization [6] that hides implementation details and adheres to a contract, usually described using a specification language (for example WSDL [3], WADL [8], Swagger [18], Apache Thrift [17] or CORBA [13]).

There is a recent trend to shift the implementation of services using the set of W3C specifications for service-oriented computing (such as SOAP and WSDL) to a lightweight approach based on the REpresentational State Transfer (REST). REST is a stateless, client-server architectural style that is being used for service-oriented computing [7]. Although REST still lacks an agreement about a language for specifying contracts, Erl et al. [5] suggest that a REST contract should at least comprise a resource identification, a protocol method, and a media type. Currently, the existing approaches for specifying contracts in REST present some limitations. For instance, Swagger specifications [18] are written in JSON (Java Script Object Notation), a general purpose notation for data representation that often leads to lengthy contracts. Swagger also does not provide any language construct for services and data type reuse. Apache Thrift provides a specification language more clear and concise, though its language is also limited with respect to both modularity and reuse, since it is not possible to specialize user defined data types (as it is possible using CORBA IDLs [13]). Furthermore, the Apache Thrift language does not present any means to extend the language used for specifying contracts.

In this paper we describe a new language— NeoIDL– for specifying REST services with their respective contracts and an extensible program generator that translates NeoIDL specifications into source code. Besides describing REST contracts in terms of resources, methods, and media types, NeoIDL specifications also include the definition of the data types used in the visible interface of a service. We considered the following requirements when designing NeoIDL. First, the language should be concise and easy to learn and understand. Second, the language should present a well-defined type system and support single inheritance of user defined data types. In addition, developers using NeoIDL should be able to specify concepts related to the *REST architectural style for service-oriented computing* [7], in order to simplify the translation of a NeoIDL specification into basic components tailored to that architectural style. Finally, both NeoIDL and the program generator should be extensible. For that reason, we designed NeoIDL to support extensibility through annotations; whereas the extensibility of the program generator relies on a pluggable architecture that uses high-order functions and some facilities present on the Glasgow Haskell Compiler (GHC) [12]. In summary, the contributions of this paper are twofold.

- We present the design and development of NeoIDL, a novel specification language for service-oriented computing that conforms to the aforementioned requirements (Section II).
- We present the implementation details of an extensible program generator written in Haskell (Section III). This contribution addresses the issue of building extensible architectures in a pure, statically typed functional language— a challenging that has not been completely discussed in the literature.

In Section IV we discuss the extensibility mechanisms and the return on investment of NeoIDL. Section V relates our contributions with existing research work available in the literature. Finally, Section VI presents final remarks and future directions of NeoIDL.

[1]DOI reference number: 10.18293/SEKE2015-218

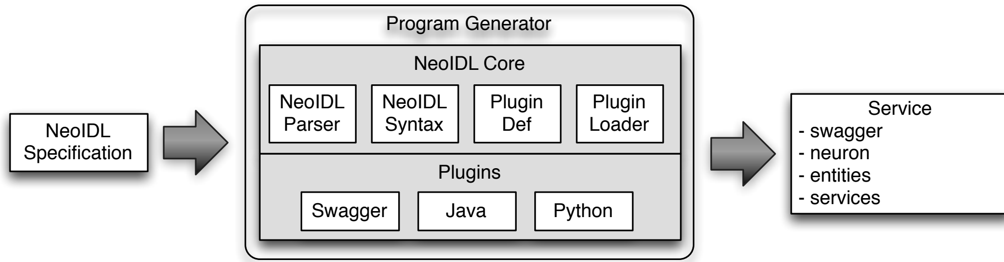

Fig. 1. Architecture of NeoIDL program generator

II. NeoIDL Design

In this section we first present an overview of our approach (Section II-A), which consists of a specification language and a program generator. Then, in Section II-B, we detail the principal constructs of NeoIDL and illustrate some examples of service specifications.

A. Approach Overview

NeoIDL has been developed to enable the specification of REST services and to allow the code generation of the implementation of those services for specific platforms. It aims to simplify the development of services, by generating code from a service specification. Figure 1 illustrates the main components of our approach, which consists of: (i) a domain-specific language (NeoIDL) for specifying REST services with their respective contracts; and (ii) a program generator that enables the code generation of REST services in different platforms. The NeoIDL generator is structured as a set of core modules, which are responsible for the parsing, syntax definition, and processing of NeoIDL specification; as well as modules for the definition and management of NeoIDL plugins. Each NeoIDL plugin defines specific extensions for the code generator that enables the generation of REST services for different platforms or programming languages.

The current implementation of NeoIDL has been already used to enable the generation of services for the `NeoCortex` platform, a proprietary framework used by the Brazilian Army. `NeoCortex` is a service oriented framework based on REST that has been developed using NodeJS— a cross-platform runtime environment for server-side and networking applications. `NeoCortex` is a polyglot framework that supports the deployment of services written in different languages (such as Python and Java) and addresses high responsiveness requirements using reactive and asynchronous programming techniques. Each `NeoCortex` service must provide a contract and a *front-controller*— which delegates a service request to the corresponding implementation. Even simple `NeoCortex` services require different components that implement business logic and other concerns, such as concurrency and persistence. In summary, a typical `NeoCortex` service comprises several components, such as:

- The `synapse` component exposes the service API using a Swagger based JSON specification, which provides an useful interface for testing a service.
- The `neuron` component implements the necessary behavior for initializing and stopping a service, as well it is responsible for mapping a requested URL pattern into a specific resource class.
- User defined data types are represented as domain classes, either in Python or Java. In the cases where it is necessary to persist a data type on a database, a database mapping is also necessary within a service.

In the context of `NeoCortex`, we translate NeoIDL specifications into Swagger specifications and other software components for different programming languages— to fulfill the polyglot requirement of `NeoCortex`. This requirement motivated us to implement the program generator of NeoIDL as a pluggable architecture (see Figure 1)— so that we are able to evolve the code generation support in a modular way. For instance, implementing a C++ program generator from NeoIDL specifications does not require any change in the existing code of the program generator. It is only necessary to implement a new NeoIDL plugin.

B. NeoIDL Language

NeoIDL simplifies service specifications by means of (a) mechanisms for modularizing and inheriting user defined data types, and (b) a concise syntax that is quite similar to the *interface description languages* of Apache Thrift and CORBA. A NeoIDL specification might be split into modules, where each module contains several definitions. In essence, a NeoIDL definition might be either a data type (using the `entity` construct) or a service describing operations that might be reached by a given pair (URI, HTTP method). Figures 2 and 3 present two NeoIDL modules: (i) the data-oriented `MessageData` module; and the service-oriented `Message` module.[2]

The `MessageData` module (Figure 2) declares an enumeration (`MessageType`), which states the two valid types of messages (a message must be either a *message sent* or a *message received*); and a data type (`Message`), which details the expected structure of a message. We use a *convention over configuration approach*, assuming that all attributes of a user defined data type are mandatory, though it is possible to specify an attribute as being optional using the syntax `<Type> <Ident> = 0;`, as exemplified by the `subject` field of the `Message` data type.

[2] The NeoIDL grammar could be found at http://goo.gl/p8eZky

```
module MessageData {
  enum MessageType { Received, Sent };

  entity Message {
    string id;
    string from;
    string to;
    string subject = 0;
    string content;
    MessageType type;
  };
}
```

Fig. 2. Message data type specified in NeoIDL

The `Message` module of Figure 3 specifies one service resource (`sentbox`). As explained, we send requests for the methods of a given resource using a specific path. In the example, the `sentbox` resource's methods are available from the relative path `/messages/sent`. This resource declares two operations: one `POST` method that might be used for sending messages and one `GET` method that might be used for listing all messages sent from a given sequential number.

Also according to our *convention over configuration* approach, we assume that the arguments of `POST` and `PUT` operations are sent in the request body, whereas arguments of `GET` operations are either sent enclosed with the request URL or enclosed with the URL path (in a similar way as `DELETE` operations). We are able to change these conventions by using specific annotations attached to an operation parameter. In these examples, conventions are used to reduce the size of services' specifications.

```
module Message {
 import MessageData;

 resource sentbox {
  path = "/messages/sent";
  @post void sendMessage(Message message);
  @get [Message] listMessages(string seq);
 };
};
```

Fig. 3. Sent message service specification in NeoIDL

To support language extensibility, NeoIDL specifications can be augmented through annotations. The main reason for introducing annotations in NeoIDL was the possibility to extend the semantics of a specification without the need to change the concrete syntax of NeoIDL. For instance, suppose that we want to express security policies for a service resource. A developer could change the concrete syntax of NeoIDL for this purpose, defining new language constructs for specifying the authentication method (based on tokens or user passwords), the cryptographic algorithm used in the resource request and response, and the role-based permissions to the resource capabilities. However, changing the concrete syntax to allow the specification of unanticipated properties of a resource often breaks the code of the program generator.

Instead, using annotations, developers might extend the language within NeoIDL specifications. Therefore, apart from the NeoIDL definitions discussed before, it is also possible to define new annotations that might be attached to the fundamental constructs of NeoIDL (i.e. `module`, `enum`, `entity`, and `resource`). Each annotation consists of a name, a target element that indicates the NeoIDL constructs the annotation might be attached to, and a list of properties. When transforming a specification, the list of annotations attached to a NeoIDL element is available to the plugins, which could consider the additional semantics during the program generation. Figure 4 presents a NeoIDL example that attaches an user defined annotation (`SecurityPolicy`) to specify security policies on the `agent` resource. In the example, using the `SecurityPolicy` annotation we specify that the operations of the `agent` resource (i) must use a basic authentication mechanism, (ii) the arguments and return values must be encoded using the AES algorithm, and (iii) only authenticated users having the *admin* role are authorized to request the resources.

```
module Agente {

 entity Agent {
   ...
 };

 annotation SecurityPolicy for resource {
   string method;
   string algorithm;
   string role;
 };

 @SecurityPolicy(method = "basic",
                 algorithm="AES",
                 role = "admin");
 resource agent {
   path = "/agent";
   @post void persistAgent(Agent agent);
 };
};
```

Fig. 4. NeoIDL specification using annotations

We end this section highlighting that the design of NeoIDL comprises a domain specific language (DSL) for specifying services APIs in a REST based environment and an extensible program generator that might evolve to generate code to different platforms and programming languages. Next section presents some details about the NeoIDL implementation, which uses Haskell as programming language— a well known language for building (embedded) DSLs [9].

III. NeoIDL Implementation

As shown in Figure 1, the implementation of NeoIDL consists of a core (split into several Haskell modules) and several plugins, one for each target language (such as Swagger, Python, or Java). The core module includes a tiny application that loads plugins definition and processes the program arguments, which specify the input NeoIDL file, the output directory, and the languages that should be generated code from the input file. Moreover, the core module contains a parser[3] and a type checker for NeoIDL specifications.

[3]We have developed the parser for NeoIDL using `BNFConverter` [16]

In the remaining of this section we present details about the implementation of two NeoIDL Haskell modules: `PluginDef` and `PluginLoader`. The first states the organization of a NeoIDL plugin and the second is responsible for loading all available plugins. The details here are particularly useful for those who want to develop extensible architectures using Haskell.

A. PluginDef component

NeoIDL plugins must comply with a few design rules that `PluginDef` states. `PluginDef` is a Haskell module that basically declares two data types (`Plugin` and `GeneratedFile`) and a type signature (`Transform = Module -> [GeneratedFile]`) defining a family of functions that map a NeoIDL module into a list of files whose contents are the results of the transformation process.

According to these design rules, each NeoIDL plugin must declare an instance of the `Plugin` data type and implement functions according to the `Transform` type signature. Moreover, the `Plugin` instance must be named as `plugin`, so that the `PluginLoader` component will be able to obtain the necessary data for executing a given plugin. Indeed, the execution of a plugin consists of applying the respective `transformation` function for a NeoIDL module, producing as result a list of files that consists of a name and a `Doc` as file content.[4]

As an alternative, we could have implemented a Haskell type class [10] exposing operations for obtaining the necessary data for a given plugin. Although this approach might seem more natural for specifying design rules for a pluggable architecture in Haskell, in the end it would lead to a cumbersome approach to our problem. The main reason for discarding this alternative approach was the need to (a) implement a data type, (b) make this data type an instance of the mentioned type class, and (c) create an instance of that data type. All those steps would be necessary for each plugin. Using our approach, the obligation of a plugin developer is just to provide an instance of the `Plugin` datatype, taking into account the name convention we mentioned above. The `language` attribute of the `Plugin` datatype is used for UI purpose only, so that the users will be able to obtain the list of available plugins and select which plugins will be used during a program generation.

B. PluginLoader component

Based on the design rules discussed in the previous section, the `PluginLoader` component is able to dynamically load the available NeoIDL plugins. This is a Haskell module (see Figure 5) that exposes the `loadPlugins` function, which returns a list with all available plugins. This list is obtained by compiling the Haskell plugin modules during the program execution and dynamically evaluating an expression that yields a list of `Plugin` datatype instances.

We assume that all Haskell modules within the top level `Plugins` directory must have a plugin definition, according to the design rules of Section III-A In Figure 5, the `loadPlugins` function lists all files within the `Plugins` directory, filters the Haskell files (files with the ``.hs'' extension), creates a qualified name to these files, and applies the `compile` function to the resulting list of qualified names. In the next step, the `compile` function uses the GHC API [12] for compiling the Haskell modules with plugin definitions and to evaluate an expression that produces a list with the available plugins.

[4]The `Doc` data type comes from the John Hughes Pretty Printer library.

```
module PluginLoader (loadPlugins) where
type HSFile = String
dir :: String
dir = "Plugins"
loadPlugins :: IO [Plugin]
loadPlugins =
  let
    pattern = isSuffixOf "hs"
    path file = dir </> file
  in (list dir) >>= (compile ∘ map path ∘ filter pattern)
dfm = defaultFatalMessager
flushOut = defaultFlushOut
compile :: [HSFile] → IO [Plugin]
compile modules =
  defaultErrorHandler dfm flushOut $ do
    result ← runGhc (Just libdir) $ do
      let hsModules = map haskellModule modules
        -- five lines of (boilerplate) code are necessary to
        -- dynamically compile Haskell code using GHC
      let exp = buildExpresson hsModules
      plugins ← compileExpr (exp ++ "::[Plugin]")
      return unsafeCoerce plugins :: [Plugin]
    return result
buildExpression :: [HSModule] → String
buildExpression hsms = "[" ++ plugins ++ "]"
  where
    plugins = concatMap (λx → x ++ ".plugin") hsms
    concat = join ","
```

Fig. 5. `PluginLoader` component

Our dynamic approach for loading plugins relies on the GHC API, using a specific idiom to compile Haskell modules and execute expressions. Figure 5 shows that idiom in the definition of the `compile` function, although we omit some boilerplate code that is necessary to compile Haskell modules using the GHC API. The last four lines of `compile` are specific to the program generator of NeoIDL. First, we build a string representation of a Haskell `list` comprising all instances of the `Plugin` datatype, obtained from the different NeoIDL plugins. Then, we evaluate this string representation of a plugin list using the *meta-programming* ability of the `compileExpr` function, which is available in the GHC API. Thus, `compileExpr` dynamically evaluates a string representation of an expression, which leads to a value that could be used by other functions of a program. The call to `compileExpr` also checks the design rule that requires (a) a `plugin` definition within all NeoIDL plugins; and (b) that definition must be an instance of the `Plugin` data type. In the cases where a plugin (exposed as a Haskell module on the top-level Plugins directory) does not comply with this design rule, a runtime error occurs. Accordingly, we use the default error handler of GHC API to report problems when loading a plugin. This is a new approach of using the GHC API to dynamically check Haskell modules in pluggable architectures.

IV. Evaluation

In this section we describe an evaluation of the NeoIDL approach through the development and generation of services in the context of a Brazilian Army project. In summary, this evaluation aims at (a) understanding the NeoIDL benefits under the ROI perspective and (b) reasoning about the modular mechanisms of NeoIDL design.

A. The use of NeoIDL in a real context

We have developed nine services that implement operations related to the domain of Command and Control (C2) [1]. These services comprehend almost 50 resources and 3000 lines of Python code. Therefore, all these services have been implemented in Python, though other projects have been implemented in Java as well.

Approximately, the number of lines of Python code related to our service repository increases according to the function $sloc = 330 \times numberOfServices$— since, in average, each service requires about 330 lines of Python code (with a standard deviation of 119). It is important to note that services are often implemented as a thin layer on top of existing components that implement reusable tasks or business logic. Accordingly, to understand the impact of NeoIDL accurately, here we do not consider lines of code related to (a) existing tasks and business logic implementations and (b) libraries that might be reused through different services.

Based on the development of these services, we estimate that it is possible to generate about 30% to 50% of a service code using NeoIDL. Indeed, in the cases that a service is *data-oriented*, involving basic operations for creating, updating, querying and deleting data, we achieve a higher degree of code generation. Differently, in the cases that a service encapsulates low level behavior (such as the implementation of a *chat-based* message protocol), we achieve a low degree of code generation using NeoIDL, mainly because the current version of NeoIDL does not provide any behavioral construct.

B. Return on Investment of NeoIDL

It is important to reason about the instant in which the design and development of a DSL pays off, since the related effort could not justify the benefits. Accordingly, here we discuss about this issue relating effort to *source lines of code* (SLOC) [15].

NeoIDL comprises almost 2500 lines of code, considering the AST code generated by `BNFConverter`. Note that nearly 67% of the Haskell code results from the `BNFConverter` parser generator. Therefore, excluding the generated code from our analysis, as well as unit testing code and make files, NeoIDL consists of 740 lines of Haskell code and 50 lines of code that (a) specifies the concrete syntax of NeoIDL and (b) serves as input to the `BNFConverter`. According to the COCOMO model [2], it is possible to compute effort from SLOC using equations (1) and (2). This leads to an effort estimation of 3.17 months, which is quite close to the real effort to implement NeoIDL, even considering that a significant effort on the design of NeoIDL was related to the successive refinements on the concrete syntax of the language.

$$\begin{aligned} personMonths &= 2.4 \times KSLOC^{1.05} \quad (1) \\ &= 2.4 \times 0.79^{1.05} \\ &= 1.87 \\ months &= 2.5 \times personMonths^{0.38} \quad (2) \\ &= 2.5 \times 1.87^{0.38} \\ &= 3.17 \end{aligned}$$

For generating the Python services to the C2 domain, the following NeoIDL modules are necessary: `bnf`, `loader`, `pluginDef`, `main`, `swaggerPlugin`, and `pythonPlugin`. These modules totalize 640 of Haskell and BNF code. Considering the discussion present in the previous section, we estimate the break-even of NeoIDL according to equations (3), (4), and (5). The third equation computes the lines of code necessary for n services without using NeoIDL; whereas the fourth and fifth equations compute the lines of code for n services, considering that NeoIDL generates 30% and 50% of the code, respectively. Therefore, the break-even of NeoIDL must be achieved after developing a number of services between 4 and 7. As a consequence, we believe that the design and development of NeoIDL improve software quality and productivity— by reducing the need to write boilerplate code, at no significant additional costs.

$$\begin{aligned} sloc &= 330 \times numberOfServices \quad (3) \\ sloc &= 0.7 \times 330 \times numberOfServices + 640 \quad (4) \\ sloc &= 0.5 \times 330 \times numberOfServices + 640 \quad (5) \end{aligned}$$

C. Modularity analysis

NeoIDL includes facilities to develop plugins and to evolve NeoIDL specifications through annotations. As explained in Section III we expose plugins according to some design rules, which allow us to develop and test plugins with a slight dependency on the existing code of the program generator. This encourages contributions to NeoIDL, by enabling developers to design and implement new plugins. In addition, it is possible to unit test a NeoIDL plugin in an isolated manner. Here we relate modularity to extensibility (it is easy to contribute to NeoIDL without a deep knowledge of the core components of NeoIDL) and testability (it is possible to test each NeoIDL plugin in isolation).

Actually, to develop a plugin, it is only necessary to understand the design rules discussed in Section III and an external library (the John Hughes and Simon Peyton Jones Pretty Printer library). For instance, Figure 6 shows the full implementation of a NeoIDL plugin, which reports basic metrics of size from a NeoIDL specification. To keep things simple, that plugin generates a file (named `metrics.data`) whose content consists of the name of a NeoIDL module followed by three lines stating the number of enums, entities, and resources within that module.

```
module Plugins.Metrics (plugin) where
import NeoIDL.Lang.AbsNeoIDL
import PluginDef
import Text.PrettyPrint.HughesPJ
plugin :: Plugin
plugin = Plugin {
  language = "Metrics",
  transformation = generateMetrics
}
print :: String → [a] → Doc
print str lst = text str <+> (text ∘ show ∘ length) lst
generateMetrics :: Transformation
generateMetrics = λ(Module (Ident s) _ _ ens ess rss) →
  let
    outputFile = GeneratedFile name content
    name = "metrics.data"
    content = vcat [text "Module" <+> text s
      , print "-enums:" ens
      , print "-entities:" ess
      , print "-resources:" rss]
  in [outputFile]
```

Fig. 6. A simple plugin for exporting metrics of a NeoIDL specification

V. Related work

Many approaches for distributed systems consider the use of an IDL, as discussed in Section I. However, similarly to CORBA [13], WSDL [3], Apache Thrift [17], and Swagger [18], the current version of NeoIDL does not support any construct for specifying formal constraints. Nevertheless, we envision that introducing the semantics of behavioral specification languages (such as Java Modeling Language [11]) into NeoIDL would (a) increase the effectiveness of program generation and (b) enable test case generation from NeoIDL specifications. It is also important to note that two shortcomings of WSDL and Swagger (lack of modularity mechanisms and low expressiveness) motivated the design of NeoIDL, which considered the syntax of other languages (CORBA, Apache Thrift) as inspiration.

Czarnecki and Eisenecker present many approaches for Generative Programming [4], including Aspect-Oriented Programming, `C++` Template Metaprogramming, and Domain Specific Languages. NeoIDL comprises a domain specific language for services' description and a pluggable architecture with an extension point that allows code generation for different target languages. Although several works describe the use of Haskell to implement (embedded) domain specific languages [9], the use of Haskell to build pluggable architectures has not been extensively discussed in the literature. Similar to the `hs-plugins` framework [14], NeoIDL architecture uses the infrastructure of the Glasgow Haskell Compiler to dynamically load and compile Haskell modules that implement NeoIDL plugins.

VI. Final remarks and future work

This paper introduced NeoIDL, a domain specific language for service specifications. We discussed the design and implementation of NeoIDL, which comprises a specification language and a pluggable architecture for generating code for different languages. We further discussed the main contributions of NeoIDL with respect to existing interface description languages (such as CORBA IDL and WSDL)— NeoIDL provides means for language extensibility and specification modularity. As a future work, we aim at writing NeoIDL plugins to generate code to other web frameworks, such as Play and Yesod Frameworks. We also intend to investigate the use of behavioral specification constructs in NeoIDL, so that we could generate test cases from NeoIDL specifications.

Acknowledgment

This work was partially supported by a research collaboration project between the Brazilian Army and the University of Brasília (project name: `GEPRO EXERCITO TDC EVOLUCAO CORTEX 2012`).

References

[1] Alberts, D.S., Hayes, R.E.: Understanding Command and Control. DoD Command and Control Research Program, 1st edn. (2006)

[2] Boehm, B.W., Clark, Horowitz, Brown, Reifer, Chulani, Madachy, R., Steece, B.: Software Cost Estimation with Cocomo II. Prentice Hall PTR, 1st edn. (2000)

[3] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (wsdl) 1.1. W3C recommendation, W3C (Feb 2001), http://www.w3.org/TR/wsdl

[4] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)

[5] Erl, T., Balasubramanian, R., Carlyle, B., Pautasso, C.: SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. Prentice Hall (2012)

[6] Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)

[7] Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (May 2002)

[8] Hadley, M.: Web application description language (wadl). W3C recommendation, W3C (Aug 2009), http://www.w3.org/Submission/wadl/

[9] Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys (CSUR) 28(4es), 196 (1996)

[10] Jones, M.P.: Functional programming with overloading and higher-order polymorphism. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming, Lecture Notes in Computer Science, vol. 925, pp. 97–136. Springer (1995)

[11] Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of jml: A behavioral interface specification language for java. Softw. Eng. Notes 31(3), 1–38 (May 2006)

[12] Marlow, S., Peyton-Jones, S.: The Glasgow Haskell Compiler. In: Brown, A., Wilson, G. (eds.) The Architecture of Open Source Applications, vol. 2. lulu.com (2012)

[13] (OMG), O.M.G.: Interface definition language 3.5. Tech. rep., Object Management Group (2014), http://www.omg.org/spec/IDL35/3.5/PDF/

[14] Pang, A., Stewart, D., Seefried, S., Chakravarty, M.M.T.: Plugging haskell in. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. pp. 10–21. Haskell '04, ACM, New York, NY, USA (2004)

[15] Park, R.: Software size measurement: A framework for counting source statements. Tech. Rep. CMU/SEI-92-TR-020 (1992)

[16] Ranta, A.: Implementing Programming Languages. An Introduction to Compilers and Interpreters. Texts in computing, College Publications (2012)

[17] Slee, M., Agarwal, A., Kwiatkowski, M.: Thrift: Scalable cross-language services implementation. Tech. rep., Facebook (2012), http://thrift.apache.org/static/files/thrift-20070401.pdf

[18] Team, S.: Swagger restful api documentation specification 1.2. Tech. rep., Wordnik (2014), https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md