International Journal of Software Engineering and Knowledge Engineering

NeoIDL: A Domain Specific Language for Specifying REST Contracts Detailed Design and Extended Evaluation

Lucas Lima, Rodrigo Bonifácio, Edna Canedo
University of Brasília
Brasília, Brazil

Thiago Mael de Castro, Ricardo Fernandes, Alisson Palmeira
Center for Development Systems of the Brazilian Army
Brasília, Brazil

Uirá Kulesza
Federal University of Rio Grande do Norte
Natal, Brazil

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

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 In addition, we also present a systematic evaluation of our approach from different perspectives, which involved the implementation of different services using NeoIDL from the domain of Command & Control. In particular, we found initial evidences that shows that NeoIDL can contribute: (i) to bring return on investment with respect to the design and development of NeoIDL, after the implementation of 4 to 7 services; and (ii) to reduce significantly the number of lines of specification when compared to an existing service specification language such as Swagger.

Keywords: Service-Oriented Computing, REpresentational State Transfer, NeoIDL, Interface Description Languages

1. 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 [10], Swagger [21], Apache Thrift [20] or CORBA [16]).

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 [8]. 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 [21] 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 [16]). 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* [8], 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) [14]. In summary, the contributions of this paper are threefold.

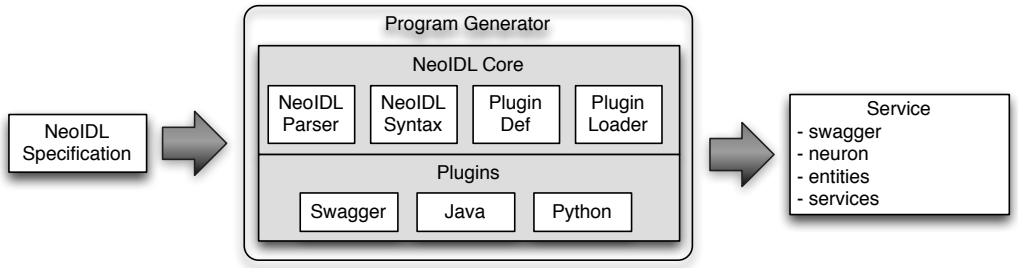

Fig. 1. Major architectural components of NeoIDL program generator

- We present the design and development of NeoIDL, a novel specification language for service-oriented computing that conforms to the aforementioned requirements (Section 2).
- We present the implementation details of an extensible program generator written in Haskell (Section 3). 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.
- - We present a systematic evaluation of our approach from different perspectives (Section 4). In our evaluation, we implemented 9 services using NeoIDL that represent operations to the domain of Command and Control (Section 4.1), which can have 30 to 50code automatically generated using our infrastructure. We present an analysis of return on investment of the usage of NeoIDL, which shows that the break-even of the language can be achieved after the development of 4 to 7 services (Section 4.2). Also a brief analysis of the modularity of NeoIDL is described by showing the amount of code to implement NeoIDL plugins for different programming languages (Section 4.3). Finally, we also compared the implementation of 44 specifications originally developed for the Brazilian Army using Swagger with their equivalent specification using NeoIDL, and we observed a reduction of 63% in terms of lines of specification considering all the contracts (Section 4.4).

Section 5 relates our contributions with existing research work available in the literature. Finally, Section 6 presents final remarks and future directions of NeoIDL.

2. NeoIDL Design

In this section we first present an overview of our approach (Section 2.1), which consists of a specification language and a program generator. Then, in Section 2.2, we detail the principal constructs of NeoIDL and illustrate some examples of service

specifications. Sections 2.3 and 2.4 present the lexical and syntactic structure of NeoIDL.

2.1. *Approach Overview*

NeoIDL has been developed to enable the specification of REST services and to allow the code generation for 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 specifications; 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 and the open-source Python Twisted Framework [7]. The `NeoCortex` platform is a proprietary framework designed by the Brazilian Army that implements a service-oriented architecture based on REST. `NeoCortex` has been developed using NodeJS—a cross-platform runtime environment for server-side and networking applications. The main reasons for developing `NeoCortex`, instead of using existing frameworks for SOC, are related to specific needs to deploy services in different platforms, ranging from well structured environments to small devices (such as tablets and mobile phones). In addition, `NeoCortex` presents a polyglot solution 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 a nutshell, a typical `NeoCortex` service comprises several components as depicted in Figure 2, which shows the internal components of a `NeoCortex` service for sending and receiving messages, including:

- 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.
- Business and utility logic that implements the expected behavior for a service and user defined data types represented as domain classes. Often, these

components are written either in Python or Java (or any other language supported by `NeoCortex`). In the cases where it is necessary to persist a data type on a database, a database mapping is also necessary within a service.

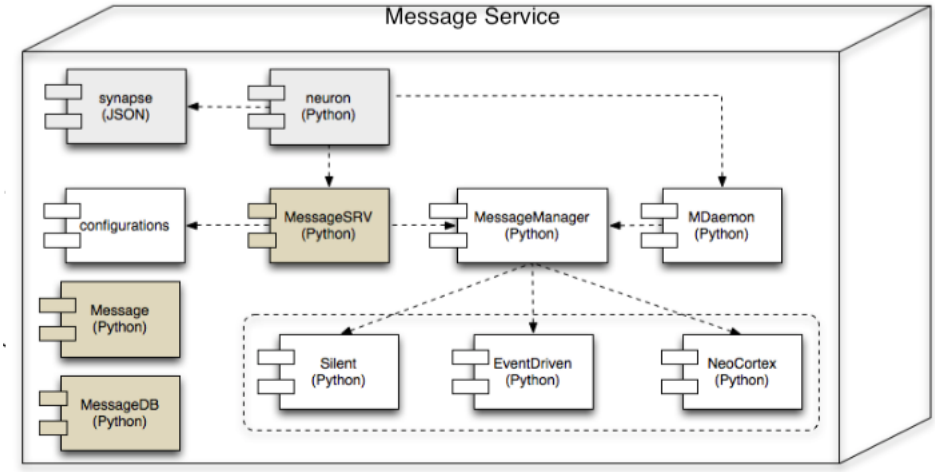

Fig. 2. Components related to a `NeoCortex` message service.

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`. Actually, NeoIDL generates code for five types of components: `synapse`, `neuron`, resource classes (`MessageSRV`), domain classes (`Message`), and database mapping (`MessageDB`). It is important to note that only the first two components should not be manually customized; whereas the remaining ones should not be overwritten by a NeoIDL translation—in the cases they have been manually edited.

The polyglot requirement of `NeoCortex` 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.

2.2. *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 or 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 3 and 4 present two NeoIDL modules: (i) the data-oriented `MessageData` module; and the service-oriented `Message` module.

The `MessageData` module (Figure 3) 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.

```
1  module MessageData {
2     enum MessageType { Received, Sent };
3
4     entity Message {
5       string id;
6       string from;
7       string to;
8       string subject = 0;
9       string content;
10      MessageType type;
11    };
12 }
```

Fig. 3. Message data type specified in NeoIDL

The `Message` module of Figure 4 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.

```
1 module Message {
2  import MessageData;
3
4  resource sentbox {
5   path = "/messages/sent";
6   @post void sendMessage(Message message);
7   @get [Message] listMessages(string seq);
8  };
9 };
```

Fig. 4. 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 5 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.

In the next sections we present the lexical and syntactic structures of NeoIDL, considering the revision of August 23, 2015. Both sections were (almost) automatically generated by the BNF-Converter [9] parser generator.

```
1  module Agente {
2
3   entity Agent {
4     ...
5   };
6
7   annotation SecurityPolicy for resource {
8     string method;
9     string algorithm;
10    string role;
11  };
12
13  @SecurityPolicy(method = "basic",
14                  algorithm="AES",
15                  role = "admin");
16  resource agent {
17    path = "/agent";
18    @post void persistAgent(Agent agent);
19  };
20 };
```

Fig. 5. NeoIDL specification using annotations

2.3. *The lexical structure of NeoIDL*

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ ', reserved words excluded.

Literals

String literals ⟨*String*⟩ have the form ‘‘x‘‘, where x is any sequence of any characters except ‘‘ unless preceded by \.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩ + ‘.’⟨*digit*⟩ + (‘*e*’-?⟨*digit*⟩+)? i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in NeoIDL are the following:

```
annotation   call      entity
enum         extends   float
for          import    int
module       path      resource
string
```

The symbols used in NeoIDL are the following:

```
{            }         ;
=            .         @
(            )         0
==           <>        >
>=           <         <=
[            ]         @get
@post        @put      @delete
/@require    /@ensure  /@invariant
/@otherwise  /**       */
*            @desc     @param
@consume     ,
```

2.4. *The syntactic structure of NeoIDL*

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the following BNF notation. All other symbols are terminals.

⟨*Modulo*⟩ ::= `module` ⟨*Ident*⟩ {
  ⟨*ListImport*⟩
  ⟨*MPath*⟩
  ⟨*ListEnum*⟩
  ⟨*ListEntity*⟩
  ⟨*ListResource*⟩
  ⟨*ListDecAnnotation*⟩
}

⟨*Import*⟩ ::= `import` ⟨*NImport*⟩ `;`

⟨*MPath*⟩ ::= ϵ
| `path` = ⟨*String*⟩ `;`

⟨*NImport*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ `.` ⟨*NImport*⟩

⟨*Entity*⟩ ::= ⟨*ListDefAnnotation*⟩ `entity` ⟨*Ident*⟩ { ⟨*ListProperty*⟩ } `;`
| ⟨*ListDefAnnotation*⟩ `entity` ⟨*Ident*⟩ `extends` ⟨*Ident*⟩ { ⟨*ListProperty*⟩ } `;`

⟨*Enum*⟩ ::= `enum` ⟨*Ident*⟩ { ⟨*ListValue*⟩ } `;`

⟨*DecAnnotation*⟩ ::= `annotation` ⟨*Ident*⟩ `for` ⟨*AnnotationType*⟩ { ⟨*ListProperty*⟩ } ;

⟨*DefAnnotation*⟩ ::= `@` ⟨*Ident*⟩ `(` ⟨*ListAssignment*⟩ `)` ;

⟨*Parameter*⟩ ::= ⟨*Type*⟩ ⟨*Ident*⟩ ⟨*Modifier*⟩

⟨*Assignment*⟩ ::= ⟨*Ident*⟩ = ⟨*Value*⟩

⟨*Modifier*⟩ ::= ϵ
| = `0`

⟨*AnnotationType*⟩ ::= `resource` | `enum` | `entity` | `module`

⟨*Resource*⟩ ::= ⟨*ListDefAnnotation*⟩ `resource` ⟨*Ident*⟩ { `path` = ⟨*String*⟩ ; ⟨*ListCapacity*⟩ } ;

⟨*Capacity*⟩ ::= ⟨*NeoDoc*⟩ ⟨*ListDefNAnnotation*⟩ ⟨*Method*⟩ ⟨*Type*⟩ ⟨*Ident*⟩ `(` ⟨*ListParameter*⟩ `)` ;

⟨*Method*⟩ ::= `@get` | `@post` | `@put` | `@delete`

2.5. *Summary of Our Approach*

We end this section by 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 [11].

3. 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 and a type checker for NeoIDL specifications. We have developed the parser for NeoIDL using `BNFConverter` [19], an easy to use parser generator that takes as input a syntax specification and generates code (both abstract syntax and parser) for different languages, including Haskell. Figure 6 represents an architectural abstraction of NeoIDL program generator.

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.

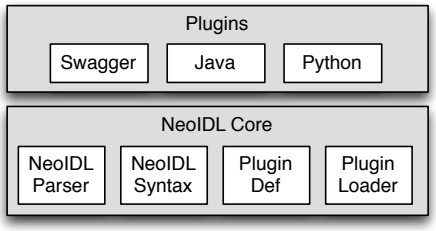

Fig. 6. Major architectural components of NeoIDL program generator.

3.1. *PluginDef component*

NeoIDL plugins must comply with a few design rules that `PluginDef` states. `PluginDef` is a Haskell module that 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.[a]

As an alternative, we could have implemented a Haskell type class [12] 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.

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

```
module PluginDef where
data Plugin = Plugin {
  language :: String,
  transformation :: Transformation
}
data GeneratedFile = GeneratedFile {
  name :: String,
  content :: Doc
}
type Transformation = Module → [GeneratedFile]
```

Fig. 7. `PluginDef` component.

3.2. *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 8) 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 3.1. In Figure 8, 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 [14] for compiling the Haskell modules with plugin definitions and to evaluate an expression that produces a list with the available plugins.

Our dynamic approach for loading plugins relies on the GHC API, using a specific idiom to compile Haskell modules and execute expressions. Figure 8 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 direc-

```
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 (\x -> x ++ ".plugin") hsms
    concat = join ","
```

Fig. 8. `PluginLoader` component.

tory) 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.

As an example of design rule violation in the NeoIDL architecture, if there is no `plugin` definition within a plugin, the following error is reported at runtime:

```
$./neoIDL
neoIDL: panic! (the 'impossible' happened)
 (GHC version 7.6.3 for x86_64-darwin):
   Not in scope: 'Plugins.Python.plugin'
```

In the last line of the above interactive section, NeoIDL program generator reports that `plugin` is not defined within the Python plugin module. In a similar

way, if the `plugin` definition is available but it is not an instance of the `Plugin` data type, a type error occurs during the execution of the NeoIDL program (see the reported message bellow, using a `plugin` definition assigned to a String value).

```
$./neoIDL
neoIDL: panic! (the 'impossible' happened)
(GHC version 7.6.3 for x86_64-darwin):
 Couldn't match expected type 'Plugin'
  with actual type '[GHC.Types.Char]'
```

Therefore, we are using GHC API to type check Haskell modules (note that Haskell is a statically typed language) during the loading phase of a NeoIDL execution. Next section presents the assessment of NeoIDL considering different goals.

4. 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 projects. We first present our experience using NeoIDL in that domain (Section 4.1). Then, in Section 4.2 we present a discussion about the return on investment (ROI) related to the design and development of NeoIDL. In Section 4.3 we present a technical evaluation about the modular design of NeoIDL. Finally, in Section 4.4, we investigate the expressiveness of NeoIDL by comparing the size of existing Swagger specifications and the size of equivalent specifications in NeoIDL. This evaluation aims at (a) understanding the NeoIDL benefits under the ROI perspective, (b) reasoning about the modular mechanisms of NeoIDL design, and (c) investigating the syntactic structure of NeoIDL.

Therefore, in the remaining of this section we answer the following questions:

- when the design of NeoIDL pays off?
- is the NeoIDL design extensible?
- what is the expressiveness of NeoIDL?

We use *source lines of code* as the metric to answer the first and third questions. We investigate the second question by means of a qualitative assessment that considers the effort to understand, test, and implement NeoIDL plugins.

4.1. *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. Here we concentrate our analysis to the services implemented in Python.

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. The boxplot of Figure 9 summarizes the SLOC distribution among the services implementations.

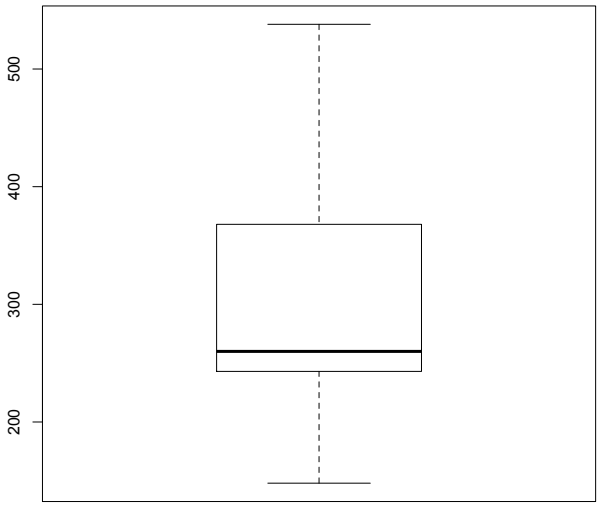

Fig. 9. SLOC distribution among services implementations

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.

4.2. *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) [18].

NeoIDL comprises almost 2500 lines of code, considering the AST code generated by `BNFConverter`. Figure 10 presents more details about the distribution of NeoIDL SLOC. 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—these refinements are not well captured within the 50 lines of code necessary to specify the concrete syntax using `BNFConverter`.

$$\begin{aligned} personMonths &= 2.4 \times KSLOC^{1.05} \\ &= 2.4 \times 0.79^{1.05} \\ &= 1.87 \end{aligned} \tag{1}$$

$$\begin{aligned} months &= 2.5 \times personMonths^{0.38} \\ &= 2.5 \times 1.87^{0.38} \\ &= 3.17 \end{aligned} \tag{2}$$

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 (see Figure 11). 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.

$$sloc = 330 \times numberOfServices \tag{3}$$

$$sloc = 0.7 \times 330 \times numberOfServices + 640 \tag{4}$$

$$sloc = 0.5 \times 330 \times numberOfServices + 640 \tag{5}$$

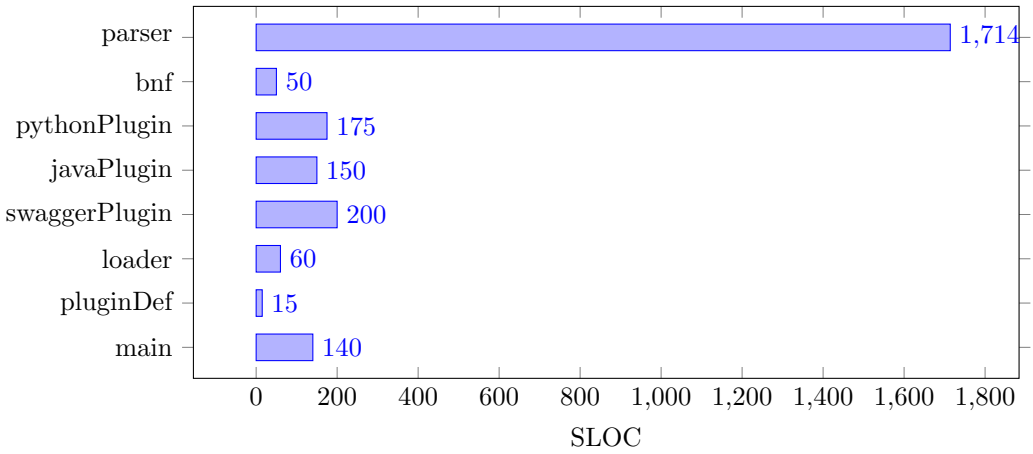

Fig. 10. NeoIDL distribution of lines of code

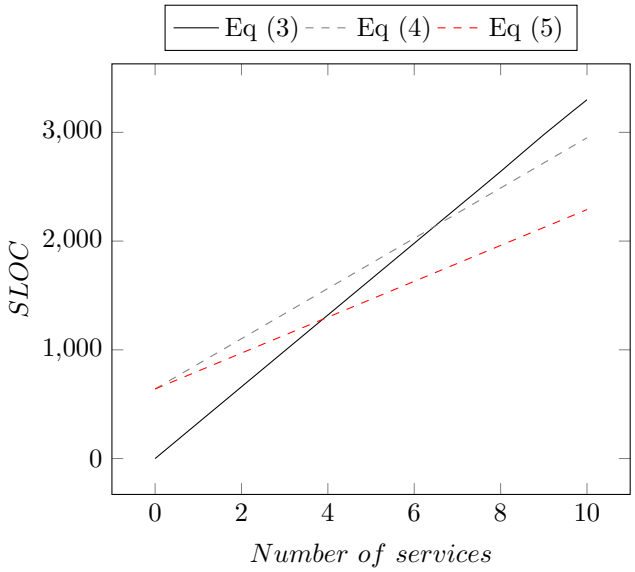

Fig. 11. Return on investment of NeoIDL

4.3. *Modularity analysis*

NeoIDL includes facilities to develop plugins and to evolve NeoIDL specifications through annotations. As explained in Section 3 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 3 and an external library (the John Hughes and Simon Peyton Jones Pretty Printer library). For instance, Figure 12 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. Thus, the `Metrics` plugin comprises only pure Haskell functions, which are independent of the technologies used to load plugins and are quite easy to test.

```
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. 12. A simple plugin for exporting metrics of a NeoIDL specification

We developed three *full-fledged* plugins: Swagger (200 SLOC), Python (175), and Java (150). Table 1 presents some metrics about these plugins. Note that both Java and Python plugins generate different types of components, while the Swagger plugin generates code for one type of component (a Swagger specification). Nevertheless, the Swagger plugin requires almost the same number of lines of code of the Java plugin. In general, to address the polyglot requirement of `neoCortex`, we expect the development of other plugins (such as C++, Haskell, and Node.js) with a level of complexity (in lines of code) similar to that found in Table 1.

In addition, based on our experience, it would not be necessary any advanced knowledge of Haskell to develop plugins, besides those shown in Figure 12.

Plugin	Generated Components	SLOC (Haskell)
Swagger	Swagger specification	200
Python	neuron services domain classes persistence	175
Java	neuron services domain classes persistence	150

Table 1. Basic information about NeoIDL plugins.

4.4. *Comparison between NeoIDL and Swagger*

In Section 4.1 we presented the comparison in SLOC between the contracts specification in NeoIDL and the resultant generated code by the framework. Another relevant issue is a comparison of expressiveness of contracts written in NeoIDL and other popular languages with the same purpose (specifying contracts for REST services). In this context, we compared NeoIDL specifications with Swagger specifications, a language that has been increasily used by the industry. Swagger [21] contracts may by writed in JSON and Yaml, both based on key-value structure.

In a collaborative work with the Brazilian Army, we obtained a portion of theirs contracts's specifications in Swagger (44 in total), specified with version 1.2. Out first step was to rewrite these specifications in NeoIDL and thus compare the total number of lines of codes (which might serve as a metric of expressiveness). The 44 contracts in Swagger comprises 13921 lines of specification, while the same set of contracts in NeoIDL comprises 5140 lines of specification. Thus, the average reduction was about 63%. In others words, it means that 10 lines of structured

Swagger specification require about 4 lines of NeoIDL specification. In this analysis we only considered *physical lines of code*, ignoring blank lines and lines consisting of delimiters only.

The reduction in number of lines is not the same in all contracts. For instance, a given service [b] required 367 lines of Swagger specification and 112 lines of NeoIDL specification. This case represents a reduction of about 69%. On the other hand, another service contract required 81 lines of specification in Swagger and 42 lines of NeoIDL specification. In this case, the SLOC decrease was slightly less than 50%.

The size of the original contract has only a small influence in the observed expressiveness. Therefore, we cannot assume that *the bigger the contract is in Swagger the bigger is the improvement (with respect to the smaller specification size) of NeoIDL*. We also realized that the use of a more descritive documentation, the number of entities, and the number of capacities do not correlate to the advantageous reduction of lines of code during a transformation of Swagger specifications into NeoIDL specifications. Therefore, it seems that the benefits does not relate to the size of the original specifications. Table 2 presents the correlation between the improvement of expressiveness (measured as the percentage of reduction obtained after transforming Swagger specifications into NeoIDL specifications) and some metrics related to the size of the original Swagger specifications.

Table 2. Correlation of the Expressiveness Improvement with the size of the Swagger specifications

Metric	Pearson's correlation	*p-value*
LOC of Swagger specification	0.19	0.20
Number of services	0.14	0.35
Number of capacities	0.14	0.34
Number of entities	0.20	0.18

Similar to NeoIDL, Swagger presents some mechanisms to reuse user defined structures. Nevertheless, this feature is almost ignored in the set of contracts we analysed, which leads to the duplication of entities' definition across different Swagger specifications. This might have occurred either due to the nonintuitive construct for reusing definitions in Swagger (based on references to JSON files) and the difficulties to identify that one entity had already been specified in another contract. After analysing the 44 Swagger specifications, we realized that 40 entities have been specified in more than one contract. Actually, one specific entity is present in 12 distinct Swagger contracts.

[b]For confidenciality reasons, the real name of contracts was omitted.

5. Related work

Our work is related to the body of knowledge on Interface Description Languages (IDLs) and Generative Programming. Regarding IDLs, as far as we know, Nestor and colleagues introduced that concept, defining interface description languages as *a formal mechanism for specifying properties of structured data*; so that it would be possible to share data among different programs [15]. According to Nestor et al., an IDL should also provide means for specifying processes to data manipulation and assertions to impose additional restrictions on a data structure. Their work also introduced many requirements that an IDL must comply, such as precision and language independence.

Many approaches for distributed systems consider the use of an IDL, as discussed in Section 1. However, similarly to CORBA [16], WSDL [3], Apache Thrift [20], and Swagger [21], 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 [13]) 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 [11], the use of Haskell to build pluggable architectures has not been extensively discussed in the literature. Similar to the `hs-plugins` framework [17], NeoIDL architecture uses the infrastructure of the Glasgow Haskell Compiler to dynamically load and compile Haskell modules that implement NeoIDL plugins. Although our approach is generic to other pluggable architectures, our implementation is specific to the NeoIDL needs.

6. 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.

References

[1] David S. Alberts and Richard E. Hayes. *Understanding Command and Control*. DoD Command and Control Research Program, 1st edition, 2006.

[2] Barry W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, Ray Madachy, and Bert Steece. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, 1st edition, 2000.

[3] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. W3C recommendation, W3C, February 2001. http://www.w3.org/TR/wsdl.

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

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

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

[7] Abe Fettig. *Twisted Network Programming Essentials*. O'Reilly Media, Inc., 2005.

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

[9] Markus Forsberg and Aarne Ranta. Bnf converter. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 94–95, New York, NY, USA, 2004. ACM.

[10] Marc Hadley. Web application description language (wadl). W3C recommendation, W3C, August 2009. http://www.w3.org/Submission/wadl/.

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

[12] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer, 1995.

[13] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *Softw. Eng. Notes*, 31(3):1–38, May 2006.

[14] Simon Marlow and Simon Peyton-Jones. The Glasgow Haskell Compiler. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications*, volume 2. lulu.com, 2012.

[15] John R Nestor, DA Lamb, and WA Wulf. *IDL-Interface description language: Formal description*. Computer Science Department, Carnegie-Mellon Univ., 1981.

[16] Object Management Group (OMG). Interface definition language 3.5. Technical report, Object Management Group, 2014. http://www.omg.org/spec/IDL35/3.5/PDF/.

[17] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging haskell in. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 10–21, New York, NY, USA, 2004. ACM.

[18] Robert Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-020, 1992.

[19] A. Ranta. *Implementing Programming Languages. An Introduction to Compilers and Interpreters*. Texts in computing. College Publications, 2012.

[20] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2012. http://thrift.apache.org/static/files/thrift-20070401.pdf.

[21] Swagger Team. Swagger restful api documentation specification 1.2. Technical report, Wordnik, 2014. https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md.