Reifier: Model-Driven Engineering of Component-Based and Service-Oriented JEE Applications

Jérôme Rocheteau[1] and David Sferruzza[2]

[1] ICAM / 35, avenue du Champ de Manœuvres, 44470 Carquefou, France
`jerome.rocheteau@icam.fr`
[2] LINA - UMR CNRS 6241 / F-44322 Nantes Cedex 3, France
`david.sferruzza@univ-nantes.fr`

Abstract. This paper aims at presenting Reifier, a tool for prototyping modules of JEE applications by the means of a model-driven development. Web services are defined as parametric components which enables to express web service patterns, to verify them formally and to reuse them in other contexts. Although Reifier requires developers to implement components compliantly to a lightweight API, it provides modelers design flexibility and verification support of their web service models.
`https://www.youtube.com/playlist?list=PLirnq05cc9voQv_9NDQrRw9za0sydYj0L`

Introduction

ICAM drives 50 applied research projects per year in various application fields either as a supplier of companies within their inner short-term projects or as partner in long-term collaborative projects. In the information technology field, projects mainly lead to develop prototypes based on web services, web technologies and mobile technologies in order to highlight innovative features. Adopted approach mostly consists in a 4-step methodology: (1) need and requirement analysis that encompasses state-of-the art and reverse engineering of existing solutions, (2) modelling, (3) prototype development and proof-of-concept, (4) skills and knowledge transfer. In fact, models are central to this approach: on the one hand, they correspond to the outputs of the two first steps and, on the other hand, they correspond to the input of the two last ones. It has been noticed that the average time spent for developing the prototype (including bug fixes) is at least twice that of spent to modelling – whereas prototyping merely consists of *transforming* models into source codes. The opposite would be better: to spend more time in modelling and to save time in verifying model implementations.

Thus, since 2014 and for its own applied research activities, ICAM began to develop a prototype, called Reifier, able to automatically generate the source code of JEE applications from a specific model of web services. A domain-specific language for web services has been designed in order to deal both with data and process models within the same model. Data model corresponds to entities with properties, inheritance and many-to-one relationships to other entities. Processes corresponds to parametric components defined by their inputs and outputs. The latter can be split, on the one hand, into atomic components, each of them linked to a computational unit manually developed and provided throughout libraries and, on the other hand, into compound components that stand for computation flows. Web services can therefore be seen as component specializations over specific entities. We argue that such a meta-model makes possible to define web service patterns which tends to reduce the number of atomic components involved in developments, and therefore the number of bugs introduced during developments, as formal verification is made possible thanks to the component-based formalization.

REIFIER's novelty thus consists in applying theorem proving techniques to web engineering and in demonstrating that it is relevant. This paper is organized as follows: The section 1 details the meta-model of web services and its associated verification operations (naming uniqueness, type system hierarchy, component consistency). The section 2 presents the programming interface required for developing atomic components that ease their verification. The section 3 explains how JEE applications are generated from such models. Finally, the section 4 compares our approach to others found in the literature.

1 Meta-Model

The meta-model of web services is defined by the means of the syntactic category model in the figure 1. It has been designed in order to match needs and requirements about verification. It does not aim to become another standard but rather to support mapping from and to web service meta-models like RAML and Swagger for instance. This category model is specified as a product type of 4 attributes: a name, a list of entities that stands for the data model, a list of components that stands for the process model and the web services that compose both entities and processes.

Firstly, entities are represented by the syntactic category entity which is defined by its name and a list of properties. An entity can optionally inherit from another entity as specified by its attribute « *entity*:entity? ». A property merely consists of its name and its type; a type being either a primitive type (string, boolean, integer, etc) or an entity. Primitive-type properties corresponds to the strict definition of entity properties whereas entity-type properties corresponds to many-to-one relationships between the entity that they belong to and the entity that they reference. This data meta-model consist in the minimal and common one among UML object-oriented data models (class diagrams), relational data models and conceptual ones (entity/association diagrams).

Secondly, components defined by the component as the sum type of abstract components or compound components. Both types of components are defined by their name, their lists of inputs and outputs and a list of parameters. Moreover, compound components own a list of concrete components. The latter correspond to abstract or compound components specialized with some valuated parame-

```
    model  =   (name:name,entities:entity*,components:component*,services:service*)

  service  =   (name:name,path:name,methods:method*)
   method  ≻   component[protocol:protocol,request:message,response:message]
  message  =   (content-type:string,content-encoding:string,headers:string*,type:type)
 protocol ::=  get | post | put | delete | head | options | trace | connect

component ::=  abstract | compound
 abstract  =   (name:name,inputs:variable*,outputs:variable*,parameters:variable*)
 concrete  =   (component:component,parameters:parameter*)
 compound  ≻   abstract[components:concrete*]

parameter  ≻   variable[term:term]
     term ::=  variable | constant
 variable  =   (name:name,type:type)
 constant  =   (type:type,value:object)
     name ::=  abstract-name(name:string)
           |   concrete-name(name:string)
           |   compound-name(names:name*)

   entity  =   (name:name,stored:boolean,entity:entity?,properties:property*)
 property  =   (name:name,type:type,required:boolean,indexed:boolean)
     type ::=  void | string | boolean | integer | float | date | entity
```

Fig. 1. Meta-Model of Web Services

ters. The name of abstract component stands for the Java qualified name of its implementation.

Finally, a web service – formalized by the syntactic category service – is defined by its name, its path and a list of methods. The syntactic category method extends that of concrete components i.e. it specializes a given abstract or compound component by valuated parameters. Such methods then consist of wrappers around concrete components with a given method protocol the signature of its required requests and that of its provided responses.

Verification Model verification focuses on names, entities and components. Verifying names consists in detecting naming collisions i.e. different elements in a given model have the same name. It is also verified that only inputs and outputs of abstract or compound components can be parametric names; neither entity, component, service nor parameter names can partially be defined by a parameter. Moreover, some naming conventions are verified: for instance, each entity (resp. component, service) name has the model name concatenated with "entities" (resp. "components", "services") as a namespace. This naming convention verification ensures that no conflict will happen during source code generation i.e. a single file will be generated twice for two different model elements. If an entity (resp. a component) does not verify such a condition, it is assumed that it points to an existing third-party entity (resp. component) and a verification is therefore performed in order to ensure that this entity (resp. this component) is provided by a library. Verifying entities consists in checking that the type system is well-founded i.e. that no cycle exists over entity inheritance relationships. Verifying components as well as methods of services consists, firstly, in checking that abstract component implementations meet their specifications (see section 2) secondly, in checking compliance between input and output types of inner components of compound components and, thirdly, in checking that every parameters of a concrete component must be declared by its inner component. It is also verified that inner component output exists within a compound component that overrides the output of one of its previous inner components. Moreover, every services must embed only components without any free variables or non-valuated parameters.

2 Programming Interface

An application programming interface (API) for developing implementations of abstract components is provided by the means of a Java library that contains:

- 2 Java interfaces: Component and Resource,
- 5 Java annotations: Parameter, Input, Output, Request and Response.

This API involves that of the Java servlets. The Resource interface merely consists in two methods setUp and tearDown able to initialize and finalize objects. The first method setUp is applied to a servlet context as parameter. This API therefore enables resource implementations to provide some *real resources* throughout servlet context sessions. Such resources can be shared between several components while initializing them.

In fact, the Component interface specifies three methods setUp, tearDown and doProcess. Whereas the first method setUp consists of initializing objects of classes that implements this interface, the second one corresponds to its dual i.e. it consists in destroying these objects. The first method is applied to a servlet context and a servlet configuration which makes possible to retrieve some resources previously inserted into the servlet context attributes. Hence resources can be shared between several components. As for the third method doProcess, it defines the computation of such components. Abstract component implementations can then process HTTP requests and responses such that it makes possible to wrap them into HTTP servlets in order to build computation flows.

The five annotations provided by the Reifier API bridge the gap between component specification and their implementations. In fact, they are used to verify that component implementations meet their specification. These annotations all concern either fields or local variables within component implementations. They are all defined by a name and a type that must comply the Reifier type system. This means that types are either primitive types or entities that are defined by qualified names of Java classes or interfaces. Types can also be references to parameters of these component implementations which type is a Java class. The latter corresponds exactly to parameter terms that are variables in the Reifier meta-model. That is why these annotations could ease verification of abstract components as Reifier type system is embedded into component implementation annotations. However, this correspond to a shallow verification as the compliance between annotations and the fields or local variables that they concern is not verified.

This API encourages developers to provide fine-grained and focused components throughout their libraries as compound component and service implementations are automatically generated from a given model as well as web service descriptions. However, it does not tackle side-effects, asynchronous method calls, first-order pre/post conditions within component implementations, as this API has been designed to be the more lightweight possible for developers.

3 Model to Text Transformation

Code generation is performed thanks to a Maven plugin and generated JEE applications rely on the REIFIER API library, Hibernate core library and on Hibernate Search library when plain text search is enabled for some indexed properties of entities (i.e. when the indexed attribute of an entity property is set to true). These libraries remains the required dependencies involved by the source code generation. It reads a XML file that describes a model compliant with the meta-model presented in the section 1. Models are then built and verified before being serialized by the means of a template engine as follows:

Firstly, entities are created as Java classes. The name of these classes is provided by the name of the entity. Private fields are declared within such classes from the entity properties, the name of fields is provided by that of properties and its Java type is defined either by Java primitive type wrappers that associated to REIFIER primitive types or by Java classes that are generated from entities. A specific field called id of type long per stored entity (i.e. those which attribute stored is set to true in a given model) is inserted that stands for the identifier or primary key of these entity occurrences. These Java fields are annotated by annotations that are provided by the Java persistence API, the Hibernate API and the Hibernate Search API accordingly. Public setters and getters for such Java fields are inserted within these Java classes. The latter do no hold neither constructor nor other specific methods. These classes correspond to data structures only. Moreover, a XML file is also created that specifies a Hibernate object-to-relational mapping between these Java classes and SQL tables from stored entities.

Secondly, compound components are created as Java classes that implements the Component interface provided by the REIFIER API. Private fields and associated setters and getters are inserted within these classes from the parameters of their components. The setUp method constructs every inner components and calls their respective setUp methods. In the same way, the doProcess method (resp. tearDown method) merely calls inner component doProcess methods (resp. tearDown methods). Moreover, every methods of services are created the same manner as Java classes that implements the Component interface; their Java class name is provided by their service name with their protocol name as suffix.

Thirdly, services are created as Java classes that extend the HttpServlet class. A private field is declared per method, the protocol name provides this field name. The Java type of such fields is defined by the generated class of these methods. Finally, the XML file web.xml that specifies, on the one hand, the previous servlet context listener and, on the other hand, both the available paths and their associated HTTP servlets is generated. This achieved the model-to-text transformation that reifies the JEE application.

In addition, a Java class that stands for a HTTP client is created per service thanks to the Apache HTTP client library. Every HTTP clients have a string field that corresponds to the server name of the generated JEE application. They all have two methods setUp and tearDown with no parameter for initializing and finalizing them. They also have a Java method per REIFIER method; its return type is defined by that of the REIFIER method response and, potentially, such Java methods have an argument defined by the REIFIER method request.

4 Related Work

This work follows the recommendation from [6, §6] that meta-model approaches are suitable for server-side code generation instead of a meta-programming one. Moreover, the meta-model presented in the figure 1 share the same concepts with that of [4]. In fact, web services are defined as pairs composed of a *process* and a *behavior*. Processes correspond either to an *action* (single-step process) or a composite one (multiple-steps process). In addition, single-step process are specified by their *behavior* that consist in inputs and outputs but also in preconditions and effects. Although our web service definition takes into account inputs and outputs, it lacks of higher order specifications as preconditions and effects.

A interesting way to embed preconditions or effects into our approach can be drawn out from [5]. In fact, a innovative web engineering methodology is designed that starts by extracting requirements from mockups i.e. quick designs of application use cases and end-user screens. Then, some refinements can be introduced into extracted models with the concept of *tags*. Tags help to specify application features such as some operations over data structures, navigation, search queries, specific actions, etc. This tag system seems lightweight for developers or modelers and this could be applied to our work through out Java annotations for atomic component implementations.

This work is highly related to that of [1] with the *Declare* tool and those from [2] to [3] with the *M3D* tool, the latter extending the previous one. In fact, *M3D* generates web application code source from a 4-layer meta-model: information layer by the means of UML class diagram, service layer by the means of BPMN, presentation layer by the means of *ad hoc* meta-model and a process layer by means of *Declare* an event constraint language based on the temporal logic LTL. Code generation is achieved using a model-to-text approach with the *Xpand* language. These tools aim at providing developers the higher flexibility possible as well as the better support possible – the less to develop and verify but the easiest to customize. This exactly is Reifier development main guideline. Our work specially focuses on the design-time support (see [1, § 3.1]) even if still requires to explicitly express services and it does not yet support formal verification. These tools have a lot in common with respect to these guidelines. However, they differ one to another at some extent. The main difference consists in the fact that *M3D* generates web applications i.e. both server-side and client-side applications whereas Reifier only generates web services i.e. server-side applications although it used to generate both sides on its earlier versions.

Conclusion

Reifier has been successfully applied to the development of several JEE applications. It has been noticed that the set of abstract component implementations reaches both quantity and quality stability over the time. It then reduces the lines of code manually written and, therefore, the number of bugs. It tends to reduce development time for server-side JEE applications and to dispatch effort differently: less for developers, more for designers.

Three main prospects are here sketched: The first prospect corresponds to an ongoing work at ICAM and consists in building other model-driven approaches on the top of Reifier thanks to model-to-model transformations. This is investigated for service-oriented management systems of sensor and actuator networks. The second and third prospects are investigated during David Sferruzza's PhD thesis. It firstly consists in turning models into parametric models in order to define explicit design patterns. It secondly consists in strengthen verification by the means of formal methods; an axiomatic semantics for components that could reach this goal has been designed.

References

1. van der Aalst, W., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - R&D 23(2), 99–113 (2009)
2. Bernardi, M.L., Cimitile, M., Di Lucca, G., Maria Maggi, F.: M3D: a tool for the Model Driven Development of Web Applications. In: Fletcher, G.H.L., Mitra, P. (eds.) Proceedings of the Twelfth International Workshop on Web Information and Data Management, WIDM 2012, Maui, HI, USA, November 02, 2012. pp. 73–80. ACM (2012)
3. Bernardi, M.L., Cimitile, M., Maggi, F.M.: Automated development of constraint-driven web applications. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. pp. 1196–1203. ACM (2016)
4. Lautenbacher, F., Bauer, B.: Creating a meta-model for semantic web service standards. pp. 376–381. INSTICC Press (2007)
5. Rivero, J.M., Grigera, J., Rossi, G., Robles Luna, E., Montero Simarro, F., Gaedke, M.: Mockup-driven development: Providing agile support for model-driven web engineering. Information & Software Technology 56(6), 670–687 (2014)
6. Scheidgen, M., Efftinge, S., Marticke, F.: Metamodeling vs Metaprogramming: A Case Study on Developing Client Libraries for REST APIs. Lecture Notes in Computer Science, vol. 9764, pp. 205–216. Springer (2016)