ABSTRACT

In this paper we present the modeling support infrastructure for domain-specific application definition. This consists of a set of meta-models and the associated generators to allow the definition of reusable and domain-specific behavior blocks, which can later be used to compose complex behaviors. In addition we also present the related visual languages that facilitate the creation of these models.

1 Introduction

The advantages of domain-specific approaches as well as the use of Domain-Specific Languages (DSLs) to effectively deal with application domains in software development have been largely discussed. In this paper we present an extension to the models presented in [6, 5, 1], that allows the modeling of high-level functions, through smaller domain-specific building blocks.

In the literature we can find many domain-specific languages, which allow to model behavior in various contexts. With regard to business process modeling, the Business Process Model and Notation (BPMN 2.0) has become the de-facto standard. With the aim at filling the Business-IT gap, significant effort has been put into bringing BPMN executable and closer to Service Oriented Architectures (SOA). While these components help alleviate agility problems that business stakeholders encounter, we observed that most of the existing solutions are domain-independent and platform-dependent, which limit the power of business matter experts at the design and monitoring stages.

The original use of the models we extend in this paper was related to moving from domain-specific descriptions to execution through transformations to Business Process Model and Notation (BPMN) [6, 5, 1], with the aim of leveraging the execution stacks of many available Business Process Management System (BPMS). However such approaches have the following drawbacks: they require technical involvement from business analysts as they must deploy the generated BPMN models into the BPMS and configure a number of parameters; The semantics of domain-specific processes don’t map directly over BPMN constructs and this entails the generation of additional activities and third party components, increasing the complexity of the resulting artifacts; they introduce a dependency to an additional language and more importantly to a full execution stack that needs to be acquired and managed separately; they don’t have mechanisms to isolate client-side code from changes on the server-side, impacting both sides during maintenance, with the disadvantages described above.

The extension proposed on this paper introduces the necessary modeling concepts so that they can be interpreted by an engine without the need for external elements. Figure shows graphically the different modeling blocks involved, as well as the relationship between them. The domain, Abstract Binding Repository (ABR) meta-model and flow meta-models are adapted and reused from the approach presented in [6]. They have been modified and updated from their original form and these changes are highlighted in the text. Their instances, the domain, ABR, and flow models respectively, are used when defining the behavior.
With the aim of illustrate the different modeling block, as well as the expressibility of them, in this paper we show how Conduit can modeled with the proposed elements. Conduit is a clone of medium.com, described in the RealWorld project \[9\]. This large open-source project is effectively a programming benchmark providing many different implementations of the same app, using different technology stacks.

A domain model can be considered as a Domain-Specific Process Modelling Language (DSPML), or in a more restricted sense, a domain-specific library. Its basic behavioral blocks, called Activities, and the corresponding domain meta-model are presented in Section 2. The ABR model contains technical configuration elements that relate to connections between the domain and a variety of services. The ABR meta-model and an example are detailed in Section 3. A flow model can be seen as a reusable function that specifies complex behavior by composing different activities defined in one or several domain models. The details of this meta-model are detailed in Section 4.

2 The Domain Meta-Model

The elements that can be modeled in a domain can be seen in Figure 2. The top-level element is Domain, and it contains a set of Activity, Service, IO and Type elements.

A data Type represents a kind of data structure. It is composed of a set of Attributes that define the type. An Attribute is defined by its id (it is a DomainObject), and its type of data, identified by a TypeRef element. A TypeRef is a reference which only points to a type, without defining it. A TypeRef can represent either a BasicType or a complex one, or indeed a set of such elements if its attribute set is true. The modeling of Types as well as the referencing mechanism have been introduced in this paper and added to the original meta-model.

As can be seen in Figure 2, the BasicTypes that can be used in a domain are: STRING, INTEGER, FLOAT, BOOLEAN, DATE, LOCATION and IMAGE.

A Service represents an entity that can be executed either by calling a function available in the system, or an external call. Note that this is an abstract service, which must be connected to the concrete implementation through the ABR model \[7\].

An IO element represents an input/output operation, i.e. information that must be retrieved from an external source or information that must be returned (for instance to be presented to a human). It is composed by a set of Variables. The IO does not specify whether the information to be retrieved must come from a human, or whether data to be exposed must be presented in a graphical user interface. For instance if the true location is requested, a mobile application could get it automatically from the device sensors, while a browser-based application could ask the end user to indicate it on a map.

Finally, an Activity represents an atomic unit of basic behaviour. As can be seen in Figure 3, the domain meta-model contains relations between Activities and the elements defined above, through a series of ServiceRelations and IORelations.

A ServiceRelation relates an Activity (it is a subtype of ActivityRelation) to a Service, and also defines a mapping between the input/outputs parameters of the service and the attributes of a Type. At runtime there is a conceptual space managed by the execution engine, where activities exchange information, called the data-flow. The activities read, create or update values from the data-flow.

In the same way, an IORelation relates an Activity to an IO element, and it also models the source of the information to publish from the data-flow (outputMappings), as well as how the data-flow is updated with the requested information (inputMappings).
The following changes have been provided in this version of Domain meta-model compared to its original incarnation:

- **Concept of IO**: This replaces a similar element called Form, that was used to model rendering policies. The new approach gives complete freedom in the representation offered to the clients of the execution element.

- **Complex type support**: The original version supports only basic types, while the new one adds multiplicity and brings increased reuse by allowing the definition of attributes to point to other domain types.

- **New basic types**: LOCATION and IMAGE were not part of the original version.

- **Data-flow mappings**: The mappings (ParameterMapping and ValueMapping) relate to data-flow access, which is a local data store. This allows more efficient data exchange between tasks in flows, compared to the alternative of persisting everything in an external database. It also allows for better monitoring support.

- **Database support**: In the original version, Entities and Relationships were specifically modeled [1], now they can be automatically derived from the Types.

Figure 4 shows a sample domain model using a graphical representation (the current implementation of the approach supports both textual and graphical modeling of domains). This corresponds to the Conduit example mentioned in Section[1] but only contains basic specific behavioral elements focused on Articles.

These elements include two types: Article that contains the actual textual information; Comment that represents an opinion about an article. Regarding Services, only one is defined, called ProcessMarkdown, used to transform a given text in Markdown format to HTML, in order to be rendered in a webpage. Some IOs can also seen in the Figure 4: ArticleIO, to obtain information about the article that a user want to post, and CommentIO to obtain comments about an article. Finally there are three activities, ComposeArticle to ask users to compose an article, ShowArticle to provide article details to the end-user, lastly ShowArticleList to retrieve and return a list of articles.
3 ABR Meta-Model

The ABR is presented in Figure 3. The Services modeled in the domain are abstract services. The Implementation meta-class is composed of implementation details as well as a set of ServiceParameters as inputs and outputs. The ServiceParameter element models the name of the real variables of the service implementation, and maps them with the Parameters.

Finally, as illustrated in Figure 4, the Implementation meta-class is abstract. The reason is that we can model various type of services, ranging from a PythonService that represents a script written in Python, to a RESTService that represents the integration with a third-party system by using Representational State Transfer (REST), to any (OtherServices). The ABR meta-model is therefore prepared to be extended with new Implementation types.

Figure 5 shows an example of an ABR model, in textual form. As can be seen, it is providing implementation details for the service ProcessMarkdown, which in this case is exposed through a POST REST call.
4 Flow Meta-Model

A Flow models a function of behavior by combining activities defined in a set of domains. As can be seen in Figure 7, it is composed of Steps and associated Transitions. On one hand, a Step represents an action that must be performed. On the other hand, a Transition (with or without conditions) establishes a relationship between source and target Steps.

Furthermore, a Transition establishes the flow of information between two Steps. This defines the data-flow, which has the effect that the information generated in the source Step will be available to be used in the target Step, and any subsequent steps as reachable through the transition graph.

The Step meta-class is abstract, therefore cannot be instantiated. There are five different types of Steps: StartStep, ActivityStep, LoopStep, ScriptStep, and DatabaseStep.

A StartStep is optionally used to indicate to the engine the execution starting point in case of possible ambiguity (that can occur in certain situation such as when some loops don’t leave out obvious first steps). Naturally, Steps of type StartStep cannot be target of any Transition.

An ActivityStep represents the usage in a flow of domain (Activities). The inputs and outputs defined in the respective activities can be set at design time, effectively overwriting any values they might otherwise hold at runtime. For instance, a variable called "user-message" could be set to a certain value by the application designer that wishes at a certain stage in the execution that a given message be displayed. To this end an ActivityStep contains a set of Overwrite elements.

LoopSteps are used to iterate over collections of variables existing in the data-flow at runtime. LoopStep is abstract, and has two sub-types, StartLoopStep and EndLoopStep modeling the beginning and end of the loop respectively. StartLoopStep is composed of the name of the set to iterate on (dataFlowSet) as well as the name of the loop in data-flow (loopName) at runtime.
A **ScriptStep** allows creation of operations such as creation and assignment of variables or addition of elements to sets. This type of **Step** can also be used to match certain functionally similar, but syntactically different variables, in cases where several compatible domains are used for the same flow.

A **DatabaseStep** is used to connect with databases. It is abstract and the available sub-types are **StoreStep**, **RetrieveStep** and **DeleteStep**. The fist one represents an operation to persist a set of variables (dataFlowVariables) existing in the data-flow at runtime. The second one is used to bring data from a database into variables in the data-flow. This **RetrieveStep** is composed of the name of the variable where the resulting data must be stored (targetVariable), the data type to retrieve (type), a boolean indicating if the resulting variable is a collection (set), as well as a set of **Criterion** used to specify the retrieval criteria. The meta-class **Criterion** represents a condition, and it is composed of a variable, an operator and a value. The last sub-type, the **DeleteStep**, represents an operation to remove data from the database, using the same selection semantics as the **RetrieveStep**. Operations to update objects from database can be performed by combining retrieve and store steps.

A flow example can be seen Figure 8 where each rectangle represents a **Step** for the execution of a domain activity, while each arrow models a **Transition**. The first step, *Get articles*, is a **RetrieveStep** and its purpose is to retrieve a set of articles stored in a database. In the next step, the articles are shown (*Show article list*) and the user select one of them, or chooses a new page listing articles (pagination), that corresponds to the activity *ShowArticleList* in the domain presented in Figure 4. Once an article is selected, its details are obtained in the **RetriveStep** (*Get article details*) and the full contents are shown to the user in the last step (*Show article*).

## 5 Conclusion and Future Work

In this paper we present an approach that allow you to define small reusable blocks of behavior called domains, that can be used to model more complex behaviors called flows.
One of the possible advantages derived of this approach is that it is possible to define a division of roles, so that people with technical knowledge can define domains, which can be used with people with less technical knowledge, to model behavior in the form of flows. Therefore is possible to empower a variety of users to create specifications of application behavior.

As future work we want to evaluate the level of real knowledge that is necessary to model behavior using flows, and thus know if people without technical knowledge can create behavior by themselves. On the other hand, we also plan to create an execution engine that can interpret flow natively.

References

[1] M. C. Cornax, J. M. Pérez-Álvarez, A. Mos, and M. T. G. López. Domain-specific data management for platform-independent process governance. In Proceedings of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th International Conference on Conceptual Modelling (ER 2017), Valencia, Spain, - November 6-9, 2017., pages 165–178, 2017.

[2] S. Kelly and J.-P. Tolvanen. Domain-specific modeling: enabling full code generation. John Wiley & Sons, 2008.

[3] T. Kosar, S. Bohra, and M. Mernik. Domain-specific languages: A systematic mapping study. Information and Software Technology, 71, 11 2015.

[4] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4):316–344, 2005.

[5] A. Mos and M. Cortes-Cornax. Business matter experts do matter: a model-driven approach for domain specific process design and monitoring. In International Conference on Business Process Management, pages 210–226. Springer, Cham, 2016.

[6] A. Mos and M. Cortes-Cornax. Generating domain-specific process studios. In Enterprise Distributed Object Computing Conference (EDOC), 2016 IEEE 20th International, pages 1–10. IEEE, 2016.

[7] A. Mos and T. Jacquin. A platform-independent mechanism for deployment of business processes using abstract services. In Proceedings of the 2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOCW ’13, pages 71–78, Washington, DC, USA, 2013. IEEE Computer Society.

[8] OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011.

[9] Y. Sun. A Real-World SPA, pages 219–246. Apress, Berkeley, CA, 2019.

[10] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. ACM Sigplan Notices, 35(6):26–36, 2000.