Towards an Infrastructure for Domain-Specific Languages in a Multi-domain Cloud Platform

Thomas Goldschmidt

ABB Corporate Research Germany
thomas.goldschmidt@de.abb.com

Abstract. Recently, cloud computing gained more and more traction, not only in fast moving domains such as private and enterprise software, but also in more traditional domains like industrial automation. However, for rolling out automation software as a service solutions to low-end, long-tail markets with thousands of small customers important aspects for cloud scalability such as easy self service for the customer are still missing. There exists a large gap between the engineering efforts required to configure an automation system and the effort automation companies and their customers can afford. At the same time, tools for implementing Domain-Specific Languages (DSLs) have recently become more and more efficient and easy to use. Tailored DSLs that make use of abstractions for the particular (sub-)domains and omitting other complexities would allow customers to handle their applications in a SaaS-oriented, self-service manner. In this paper, we present an approach towards a model-based infrastructure for engineering languages for a multi-domain automation cloud platform that make use of modern DSL frameworks. This will allow automation SaaS providers to rapidly design sub-domain specific engineering tools based on a common platform. End-customers can then use these tailored languages to engineer their specific applications in an efficient manner.

1 Introduction

Recently, cloud computing gained more and more traction not only in fast moving domains such as private and enterprise software but also in more traditional domains like industrial automation. For example, offering automation software as Software-as-a-Service (SaaS) allows companies to reach customers that could not afford to maintain a complete on-premise system automation. The type of automation software which this targets is level 3 (defines the activities of the work flow to produce the desired end-products) and selected areas of level 2 (defines the activities of monitoring and controlling the physical processes) of the ISA-95 standard [7], i.e., activities of manufacturing execution systems (MES). Typical sub-domains for this low end automation are building automation or different areas in the smart grid domain such as renewable power generation and electronic vehicle charging. Furthermore, with the advent of the Internet of Things (IoT) traditional automation tasks such as monitoring and control also broaden their scope to more and more smaller and privately deployed applications.

J. Cabot and J. Rubin (Eds.): ECMFA 2014, LNCS 8569, pp. 242–253, 2014.

However, in these low-end markets there exists a large gap between the engineering efforts required to configure an automation system and the effort automation companies and their customers can afford. On one hand, automation companies cannot offer engineering to tens of thousands of customers which are the target for their automation SaaS. Therefore, a prerequisite for SaaS to work on a large scale is that customers can do self-service on their applications, i.e., do engineering tasks on their own. On the other hand such customers do not have the expertise and cannot afford the expenses for engineering the automation system themselves based on the current complexity of engineering. Current generic engineering languages such as the languages defined by the IEC 61131-3 standard are designed for expert automation engineers and are thus often too complex for non-experts. Furthermore, for most of the tasks in these low-end domains, where an automation SaaS solution is a good fit, such complex capabilities are not needed. A tailored Domain-Specific Language (DSL) [4] that makes use of abstractions for the particular (sub-)domain and omitting other complexities would allow customers to handle their applications in a SaaS-oriented, self-service manner.

At the same time, tools for implementing domain-specific languages have recently become more and more efficient and easy to use. For example, for the embedded systems domain an extensible platform called *mbeddr* [15] was created that allows for extensibility and modularity [13] based on an underlying base language, which is based on C. Such platforms allow to tailor languages to specific needs in sub-domains or even individual projects. A main advantage of modern DSL systems such as MPS [8](on which mbeddr is based) is that the DSL development environment and the runtime environment for the created languages are based on common IDE. This allows for rapid prototyping and development of the DSLs and the editors for the DSLs. Created languages can be tested and used on the spot and do not require an extensive generation and compilation procedure. Additionally, the coupling between the metamodel and views on that metamodel becomes more and more loose. For example, MPS now allows to have multiple textual, graphical or tabular view types on the same metamodel. Using these mechanisms, a DSL for the specific sub-domain can be produced based on the needs of the intended users.

In this paper, we present an approach towards a model-based infrastructure for engineering languages for a multi-domain automation cloud platform. A central model repository based on an industry standard information model (OPC UA [10]) serves as storage for all metamodels, language definitions and engineering models. A web-enabled DSL framework (MPS [8]) provides language engineering functionality on top of these models. Finally, the platform supports different roles in a domain's ecosystem: domain expert, language engineer, domain-specific engineer and operator. Hence, the complete life-cycle of a DSL ands its corresponding engineering system is supported in the platform, starting from the definition of the domain's metamodel, the language definition, up to the use of the language by engineers and its runtime environment. We implemented a proof of concept prototype showing the technical feasibility of the system.

The contribution of this paper is twofold. First, by presenting our envisioned approach we give practitioners a basis on how domain-specific engineering on scalable, multi-domain cloud platform can be realized. Second, we raise conceptual and technical challenges that we identified to the attention researchers to provide indications for future research.

This paper is structured as follows. Section 2 gives a background on OPC UA [10] which is one of the main concept/technology used in our platform. In Section 3 we present the conceptual architecture of the platform. We give an overview of the proof-of-concept prototype we implemented in Section 4. Conceptual and technical challenges encountered are presented in Section 5. Finally, Section 6 summarizes related work and Section 7 concludes and presents future work.

2 OPC Unified Architecture

OPC UA is a well established industrial standard in the automation domain for communication as well as for making information models accessible. As it already provides basic building blocks for describing runtime systems as well as the domain-specific engineering artifacts (e.g., control programs, device configurations, etc.) it is a good candidate to serve as technical foundation for the common information model within our platform. This section explains the technical features of OPC UA and tries to highlight parts that might still be missing to work in our platform.

OPC UA provides a secure, reliable, high-performance communication infrastructure to exchange different types of data in industrial automation. That includes current data like measurements (e.g. from a temperature sensor) and setpoints (e.g. for defining the desired level in a tank), events (e.g. device lost connection) and alarms for abnormal conditions (e.g. a boiler reached a critical level). In addition, it provides the history of current data (e.g. the temperature trend for the previous day or the last ten years) and of events (what events of a certain type occurred the last five days). In order to provide semantic with the data, also meta data is exchanged in terms of an information model. In Figure 1 depicts an example of an OPC UA address space using the standard graphical representation defined by the OPC UA specification [11].

On the right hand side in Figure 1 the type system is shown, with object types in a type hierarchy. For example, the *DeviceType* is an abstract object type representing all kinds of devices. It defines a variable called *SerialNumber*. A subtype *TemperatureSensorType* adds the *Temperature* variable, including the *EngineeringUnits*. Variables are typed as well, like the *Temperature* of type *AnalogItemType* defined by the OPC Foundation. This type adds a property to the variable containing the *EngineeringUnits*. On the left hand side in Figure 1 an instance of the *TemperatureSensorType*, *TempSensor1*, is shown. The instances contain the concrete values, like the temperature measured by *TempSensor1*.

OPC UA is based on a client server model where the client asks for data and the server delivers the data. The client has the option to read and write the data,

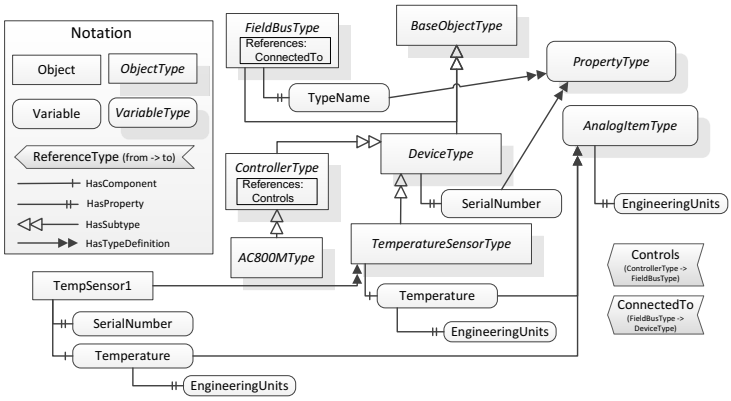

Fig. 1. Example of an Address Space in OPC UA

but also to subscribe to data changes or event notifications. In addition, the client can browse the address space of the server and read the meta data information. For large and complex address spaces the client also has the capability to query the address space for information, for example asking for all temperature sensors that are currently measuring a temperature larger than 25 °C.

The most prominent set of engineering languages, namely the 5 languages defined by the IEC 61131-3 standard, has an official OPC UA representation defined by PLCOpen [12]. The standard defines different domain-specific languages, i.e., Structured Text, Instruction Lists, Function Block Diagrams, Ladder Diagrams and Sequential Function Charts. The languages are partially interchangeable and overlap to a certain extent. For example, the Function Block Diagrams language can be used to connect and orchestrate existing executable code blocks which, in turn, can be implemented by the Structured Text language.

Based on the function block (CTU_INT) and program code (MyTestProgram) in Listings 1.1 and 1.2 the corresponding OPC UA representation is given in Figure 2. Using this kind of information model the programs running in a PLC can be monitored just as any other variable. Generic, OPC UA based program visualizations can then be used to monitor the state of programs using the same way as for their primary variables. This eases the maintenance of the programs and helps engineers, for example, during debugging.

However, the current specification of the PLCopen OPC UA representation does not include the executable parts of the function blocks and programs. For example, the OPC UA representation in Figure 2 shows that it only includes the variables (e.g., CU, R, PV) specified in Listings 1.1 and 1.2 but not the dynamic code parts such as if-then-else blocks. For a complete representation of the control programs in OPC UA this would be required.

3 Conceptual Architecture

The proposed common platform can provide basic automation functionality such as acquiring data from the field as well as storing, analyzing and visualizing it.

Domain-specific extensions can then focus on special protocols for communication, algorithms for data analysis for the particular semantics of the data, or approaches for doing control or solving optimization problems within that domain. Targeting each of the many automation sub-domains with a specific SaaS solution would cause massive development and maintenance efforts within automation companies. Therefore, a common automation cloud platform helps to focus development on domain-specific engineering tools and applications on top of a common infrastructure.

The main requirements that motivate the architecture of our platform are (a) easy and fast creation of languages for new sub-domains, (b) scalability to hundreds and thousands of parallel users, (c) compatibility with existing communication technologies in the automation domain, (d) 3rd party extensibility for the creation of new languages.

```
1  FUNCTION_BLOCK CTU_INT
2  VAR_INPUT
3      CU: BOOL;
4      R: BOOL;
5      PV: INT;
6  END_VAR
7  VAR
8      PVmax: INT := 32767;
9      CU_OLD: BOOL;
10 END_VAR
11 VAR_OUTPUT
12     CU: BOOL;
13     Q: BOOL;
14     CV: INT;
15 END_VAR
16     IF R THEN
17         CV := 0;
18     ELSEIF ((NOT CU_OLD)
19         AND CU
20         AND (CV < PVmax)) THEN
21             CV := CV + 1;
22     END_IF;
23     Q := (CV >= PV);
24     CU_OLD := CU;
25 END_FUNCTION_BLOCK
```

Listing 1.1. Example function block implementation.

```
1  PROGRAM MyTestProgram
2  VAR_INPUT
3      Signal: BOOL;
4  END_VAR
5  VAR
6      MyCounter: CTU_INT;
7  END_VAR
8  VAR_TEMP
9      QTemp: BOOL;
10     CVTemp: INT;
11 END_VAR
12     MyCounter(CU := Signal,
13         R:= FALSE, PV := 24);
14     QTemp := MyCounter.Q;
15     CVTemp := MyCounter.CV;
16 END_PROGRAM
```

Listing 1.2. Example 61131-3 program.

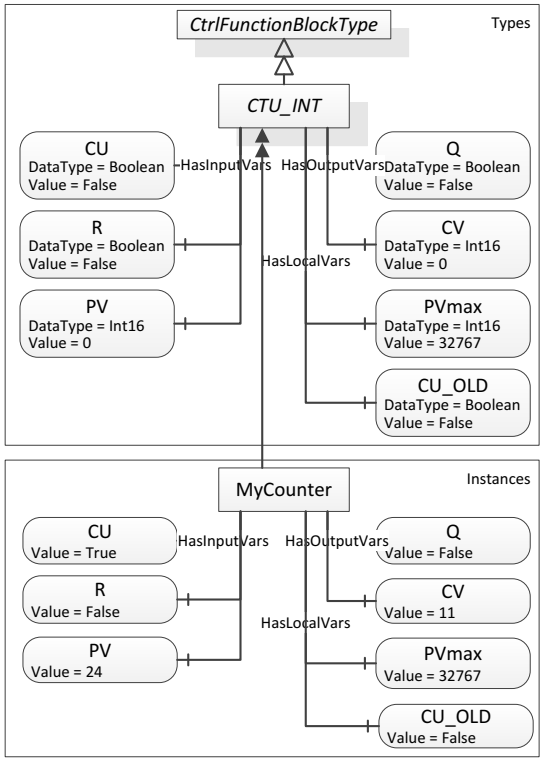

Fig. 2. Example model representation in OPC UA

By providing customizable domain-specific engineering languages as a building block we facilitate self-service engineering and enable the platform to scale to a large number of customers. A DSL infrastructure that has a common way of

creating and using languages as well as engineering, accessing the artifacts created with the language can form this building block. To achieve this we propose a model-based approach based on a common model repository that is deployed in the automation cloud platform. Figure 3 gives an overview on how we envision our model-based DSL infrastructure for a multi-domain automation cloud platform could look like.

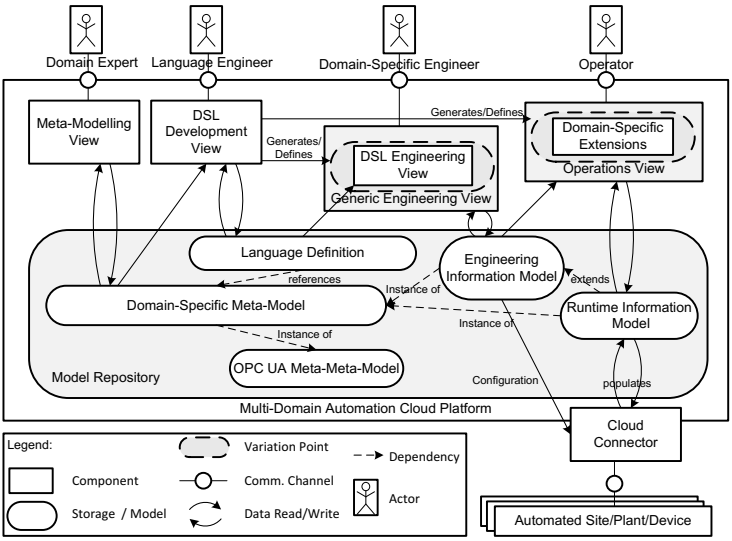

Fig. 3. Architecture of the language and modeling environment of a multi-domain automation cloud platform

Model Repository: A main component of the infrastructure is the common model repository that hold, metamodels, language definitions, engineering models as well as runtime information models. As introduced in Section 2, OPC UA provides a meta-metamodel tailored for the automation domain, as well as a common way of accessing all models on all meta levels. Furthermore, OPC UA is prepared to serve as a basis for building DSLs on top of it [5]. These capabilities make OPC UA a good foundation for the common model repository. Additionally, OPC UA does not imply a storage format for the models but rather defines the way on how models are exposed. Therefore, we combine an OPC UA information model access layer with a scalable cloud database (such as NoSQL databases). This combination provides a scalable, multi-tenant model repository for our system.

Furthermore, OPC UA specifies a query interface for browsing and finding nodes within the information model but does not prescribe the underlying query implementation. Thus, it is possible to map model queries to the underlying persistence technology and its efficient query mechanisms.

As all models are then available in the OPC UA address space and have the corresponding links between each other a cross-model navigation can easily be

implemented. Furthermore, OPC UA allows platform developers to store, browse, and manage all models the same way. Thus, making it easy to add generic services for all the above phases. For example, versioning services or social/community sharing services can be built that work for all created languages at once.

Web-based Editors: Modern web-based editors[1] can be used as an efficient way to interact with an online development system. Therefore, all user interaction with our system (domain meta-modeling, language engineering, domain-specific engineering, operations) can be implemented as a web-based editor directly working on the online model repository.

A fundamental principle that we follow for our editors is that they present views on a common model. Thus, it is possible to use different types of views on the same model. Different types of users can then use specifically tailored views to interact with their models. A prerequisite for the use of this view-based approach is a clear separation between the languages and their editors which represent the different views types and the underlying model.

Domain Meta-Modeling: As depicted in Figure 3, using the meta-modeling view domain experts can then define the metamodel of their domain. The meta-modeling view can be implemented in different ways. For domain experts familiar with OPC UA a generic graphical OPC UA editor can be used. However, being a view-based approach we also plan to integrate other views that suite different domains, such as a UML-like view for experts that are nearer to software engineering. An informal mapping from UML to OPC UA is already defined in the OPC UA specification [11]. Furthermore, textual views for defining metamodels can be added.

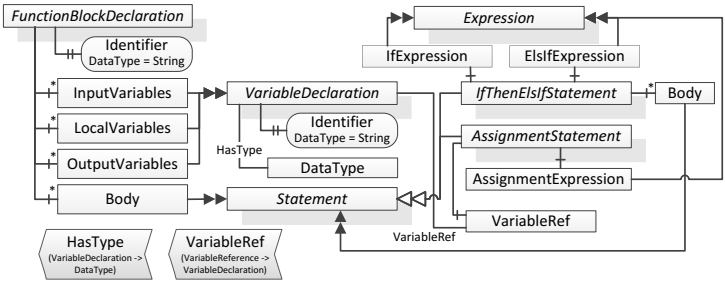

Fig. 4. Excerpt from a metamodel defined using OPC UA for the IEC 61131-3 languages

Figure 4 depicts an example metamodel defined using OPC UA. It shows an excerpt of different concepts of the IEC 61131-3 specification, such as a function block that has input and output variables but also the actual control algorithm (body having statements) can be represented in that way.

Language Engineering: Based on the domain-specific metamodel, language engineers use the DSL development view to define one or more languages that

[1] E.g., Cloud9 IDE (`https://c9.io/`)

implement the views that are tailored for the targeted domain-specific engineers' needs. The resulting editor can then be plugged into the engineering view extending it with domain-specific capabilities.

To achieve a representation of languages on top of our common model repository we require a DSL framework that has an interchangeable storage layer. Furthermore, the language specification metamodel (having constructs like *language specifications, view type definitions, templates,* etc.) needs to be mappable to our OPC UA based information model. Finally, the generated editors have to be web-enabled so that we can run them on our cloud platform.

Additionally, we envision the DSL infrastructure to support extensible views so that tailored domain-specific languages can be created on top of base languages that are executable (such as Java or a 61131-3 language where we have a cloud based interpreter). An example extension language, a simple cause and effect matrix editor for the building automation domain could be created as a tabular view on the domain-specific metamodel. E.g., associating light switches with room lights can be easily mapped by selecting the appropriate cells in the matrix. A mapping of such matrices to 61131-3 was introduced in [1] and could be implemented for this editor as well.

Domain-Specific Engineering: The role of the domain-specific engineer can now be taken over in self-service by the SaaS customer. Such engineers will then develop and configure the customer specific applications which in turn is also stored as an instance (engineering information model) of the domain-specific metamodel. This engineering model is also input to the cloud connector component which is responsible for handling the data coming from or going to the automated site, plant or device(s). This data then populates the runtime information model which is the basis for functionality such as history, analysis, control and visualization. Finally, operators can use the operations view, including domain specific extensions also coming from the DSL to interact with the system. Figure 5 shows an editor that we created based on the OPC UA metamodel given in Figure 4. It defines a textual view type for the function blocks based on the syntax definition given in the IEC 61131-3 specification.

Runtime System: Other components, for history, analysis and execution control algorithms are also part of the automation cloud system but are out of scope of this paper. However, it is important to note that all real-time critical control software will have to remain local to the plant. Only higher level control and optimization task with longer cycle times (e.g., greater than a second) will be part of the automation SaaS.

4 Prototype

Based on a survey we executed earlier[2], which was based on a tool-oriented taxonomy of view-based modeling [6], we analyzed different technologies for implementing a proof-of-concept prototype for our platform. We needed a tool that

[2] An earlier but published version of this is available here: `http://sdqweb.ipd.kit.edu/burger/mod2012/`

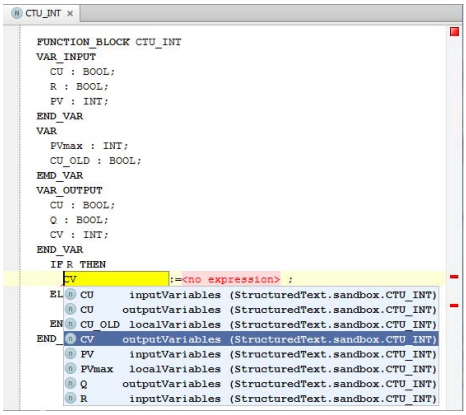

Fig. 5. Editor created for the example metamodel defined in Figure 4

supports both textual as well as graphical syntaxes in a projectional, view-based manner. Furthermore, we selected projectional partiality (allowing different view types to work on different parts of a metamodel), view overlap as well as intra/inter view type overlap (allowing different views at the same time on the model as well as different view types on the same model) as main required features. In addition to the selection properties from the taxonomy, the tool shall support web-based views and have an exchangeable storage layer.

The JetBrains Meta Programming System (MPS) [8] is an approach for the creation of textual modeling languages. MPS provides the possibility to internally map the language to Java where it can then be executed just like an internal DSL. However, MPS also allows to define a mapping to other base languages. The biggest difference to most other textual modeling language approaches is that MPS persists a kind of Abstract Syntax Tree (AST) of the language's instances instead of persisting the concrete syntax representation as text file. The editors manipulating the AST are projectional editors that create the textual representation on the fly. For upcoming versions (3.1) also graphical representations are be supported. The use of the AST as the main underlying model allows for the use of multiple, alternative concrete representations of a model. These representations may also project only a certain part of the underlying model to the concrete syntax. The projections created based on the AST are not persisted, thus MPS does not support custom formatting.

Currently, to the best of our knowledge MPS [8] fulfills or will eventually fulfills our previously posed requirements. It supports textual and tabular views already and is on the way of supporting graphical views [3]. Furthermore, web-based views are also supported[4]. Therefore, our current proof-of-concept prototype is based on this technology. Another important feature which we require for our platform,

[3] MPS [3] roadmap: Q1 2014: MPS 3.1 Support for diagrams in editor

[4] Early version available here: `https://github.com/JetBrains/jetpad-projectional`; Roadmap Q2 2014: MPS 3.5 Web-based projectional editor

i.e., the exchangeable storage layer, is also supported by MPS as it supports custom persistence.

As mentioned earlier we envision our platform to be based on a common OPC UA based model repository. We implemented a prototypical MPS persistence provider for OPC UA. Additionally, we created an implementation that allows us to store OPC UA models in a cloud based database. Furthermore, we started implementing a set of editors for the 61131-3 languages based on MPS. The example editor presented in Figure 5 is one of these editors.

5 Conceptual and Technical Challenges

We currently see a several conceptual and technical challenges on our way towards the envisioned platform. Some of them are specific technical questions where we aim to extend our proof-of-concept prototype to evaluate them. Others, are on a conceptual level where more and broader research would be required. The challenges we see so far are given in the list below. We do not deem this list as being a complete picture, new challenges might arise over time as we further develop the platform.

(1) The mapping between the abstract syntax tree (AST) as defined by MPS and OPC UA has to be validated more extensively. Can all constructs used in the DLS tool be represented in OPC UA. Is reference handling done in a consistent way? (2) A great advantage of online editors is concurrent model editing. Meaning, multiple users can work in parallel on the same content and receive immediate updates from one another. For collaborative engineering this might give real benefits regarding the engineering efficiency. Cloud applications such as Google Drive already nicely support this feature for office documents. However, it remains to be evaluated how big these benefits really are and how good the combination of an MPS-based web-editor and the underlying OPC UA-based persistency support this kind of feature. (3) The usability of the web-editors may be a crucial point for the acceptance of the platform to a large number of customers. Therefore, we plan to employ metrics that assess the usability of the editors for different types of users. (4) Another challenge we see is the validation of the created languages. How can we ensure that the languages and abstract view types that are developed, also by 3rd parties can be mapped to the underlying execution engines correctly. We would need to do verification on the language definitions and the transformations to ensure this. However, there exists limited related work in this area which could serve as a basis for this task.

6 Related Work

A multitude of approaches for the creation of domain-specific languages exists. To mention a few of them: The Graphical Modelling Framework (GMF) [2], which is part of the Eclipse Modelling Project provides means for creating graphical modeling languages based on Ecore metamodels. Language engineers may specify which elements of a metamodel should be editable through a specific

diagram. This allows for projectional view types. MetaEdit+ [14] is a commercial tool for creating graphical domain specific modeling languages. Support for the integration of multiple languages has also been investigated using this approach. This also enables the approach for multi view type modeling, as different languages may cover different parts of an interconnected, common metamodel.

Spoofax [9] is a language workbench based on scannerless parser generator Stratego/SDF. Due to the scannerless parsing mechanism it features extensive support for modularization of languages. However, explicit support for view-based modeling, is not given in this approach. There will be still one main view type, i.e., the textual one that needs to be complete. Other, additional view types then may be partial and also have a different representation.Xtext [3], is the official textual modeling approach of the Eclipse Modelling Project. Its primary use case is the integrated definition of concrete and abstract syntax based on a grammar-like specification. Additionally existing metamodels may be imported and and enriched with a view type definition. A language engineer may define different syntax elements for the same class allowing for intra view type overlaps.

However, none of the above mentioned tools provide support for textual as well as graphical projectional views and web-based editors at the same time. Only this would allow for a deployment on a cloud platform. For us this justifies the choice of MPS as a core technology in our platform. Regarding a cloud-based DSL platform very little related work exists. Cloud based editors, such as Cloud9 IDE (`https://c9.io`) or WriteLatex (`http://writelatex.com`) provide online editors for specific languages. However, they do not allow their tenants to define own languages and can therefore not be considered a complete cloud-based DSL infrastructure.

7 Conclusions and Future Work

In this paper, we presented a conceptual architecture for a multi-domain engineering cloud platform for the automation domain. The platform is based on a central model repository implemented on top of the OPC UA meta-metamodel and supports the entire life-cycle of a domain-specific language from metamodel definition to operations of the engineered system. Furthermore, we raised conceptual and technical challenges we encountered that give researchers hints for future research.

Based on the proposed architecture we aim to complete our proof-of-concept implementation that facilitates a combination of an OPC UA based model repository, MPS [8] as language engineering workbench and web-based views for the different roles. We plan to use the building automation domain as case study to implement domain-specific languages and the corresponding protocol connectors (e.g., KNX and EnOcean) for the cloud connector. Furthermore, we intend to build a full-fledged IEC 61131-3 web-based editor based on our existing proof-of-concept prototype to evaluate if and how a complex system of languages can be built on top of the proposed infrastructure.

References

1. Drath, R., Fay, A., Schmidberger, T.: Computer-aided design and implementation of interlock control code. In: 2006 IEEE Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, pp. 2653–2658 (October 2006)
2. Eclipse Foundation. Graphical Modeling Project (GMP) Homepage, `http://www.eclipse.org/modeling/gmp/` (last retrieved December 17, 2013)
3. Eclipse Foundation. Xtext Homepage, `http://www.eclipse.org/Xtext/` (last retrieved January 08, 2014)
4. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010) ISBN 0321712943
5. Goldschmidt, T., Mahnke, W.: Evaluating domain-specific languages for the development of OPC UA based applications. In: 7th Vienna International Conference on Mathematical Modelling, Special Session Modelling and Model Transformation in Automation Technologies, MATHMOD (2012)
6. Goldschmidt, T., Becker, S., Burger, E.: Towards a tool-oriented taxonomy of view-based modelling. In: Modellierung, pp. 59–74 (2012)
7. ISA. International standard for the integration of enterprise and control systems, `http://www.isa-95.com/`
8. JetBrains. Meta programming system - DSL development environment (2013), `http://www.jetbrains.com/mps/`
9. Kats, L.C., Visser, E.: The spoofax language workbench: Rules for declarative specification of languages and ides. In: ACM Sigplan Notices, vol. 45, pp. 444–463. ACM (2010)
10. Mahnke, W., Leitner, S.H., Damm, M.: OPC Unified Architecture. Springer Press (2009)
11. OPC Foundation. OPC UA Specification: Part 3 - Address Space Model, `http://opcfoundation.org/UA/Part3` (2010)
12. PLCopen and OPC Foundation. Explanation of the combined technologies of plcopen and opc foundation (2009)
13. Ratiu, D., Voelter, M., Molotnikov, Z., Schaetz, B.: Implementing modular domain specific languages and analyses. In: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa 2012, pp. 35–40. ACM, New York (2012) ISBN 978-1-4503-1801-3
14. Tolvanen, J.-P., Kelly, S.: Metaedit+: Defining and using integrated domain-specific modeling languages. In: Proceeding of OOPSLA 2009, pp. 819–820 (2009) ISBN 978-1-60558-768-4
15. Voelter, M., Ratiu, D., Kolb, B., Schaetz, B.: mbeddr: Instantiating a language workbench in the embedded software domain. Automated Software Engineering 20(3), 339–390 (2013)