Implementation of “smart greenhouse” visual programming tool using deep metamodeling

Elizaveta Kuzmina
Saint Petersburg State University
Saint Petersburg, Russia
Email: kuzminaeliz@gmail.com

Yurii Litvinov
Saint Petersburg State University
Saint Petersburg, Russia
Email: y.litvinov@spbu.ru

***Abstract*—DSM platforms that are based on metamodeling typically use the two-metalevel approach. Disadvantages of this approach were taken into account when developing the REAL.NET platform, and deep metamodeling approach was chosen. The following article describes an experiment to create a “smart greenhouse” programming technology on the basis of REAL.NET. The experiment has shown the efficiency of the platform for quick creation of tools for the end user programming. The article describes the platform itself and deep metamodeling approach as well as “smart greenhouse” programming technology based on it.**

***Index Terms*—Domain-specific modeling, visual languages, multi-level metamodeling**

1. Introduction

The days when visual modeling was considered a new silver bullet are long gone, but visual modeling is still a viable tool for end-user programming. Non-technical people often don’t have time to study even a simple programming language, and in scenarios where some simple programming is needed visual languages can be very effective alternative. We believe that number of applications of visual languages for end-user programming will grow with the adoption of Internet of Things, blockchain smart contracts, simplification of mobile application and web application development and so on.

Growth of visual end-user programming technologies is hindered by a very high cost of tool development and usability issues. Adequate tools for visual languages are much harder to develop than textual IDEs due to complexity of graphical editing features required, and when a tool is needed only to perform a small set of tasks for a limited amount of users, costs of its development are much higher than expected benefits. To address this, a considerable amount of research was done on creating tools that simplify visual modeling tools development — Domain-Specific Modeling platforms (or DSM platforms).

The core of every DSM platform is its ability to declaratively specify a visual language and automatically provide tools like visual editors, code generators, model browsers and so on. Many different approaches and formalisms were developed for language specification, most of them are using metamodels — models (visual or textual) that specify a set of valid models that becomes a new language. Metamodel is itself a model created using dedicated metalanguage, much like Backus-Naur form describing the grammar of textual language. There are several existing metamodel architectures, most widely known being two-level architecture (with dedicated metalanguage that allows to specify needed visual language), used, for example in MOF (Meta-Object Facility, metalanguage with which UML is specified) and several popular DSM platforms, like Eclipse Modeling Project [1] and MetaEdit+ [2].

Limitations of two-level metamodeling architecture became apparent when UML 2 standard was discussed (see [3], [4]), and new metamodeling architectures have emerged with the aim to simplify language definition and improve tool support. For example, UML is not able to capture instance-of relation between classes and objects on a language level, as “Class” and “Object” are different instances of the same element of metalanguage and are not related to each other. So every UML tool needs to have some complicated custom code to maintain consistency of models that use classes and objects — for example, that every object in a model has its corresponding class somewhere. One of the most developed architectures that allows to solve such problems is deep metamodeling [4], [5]. Deep metamodeling proposes to consider entities of a model as classes and objects at the same time (and call them “clabjects”). Clabjects can be used in a model and at the same time be used as types for a lower-level model, for example, clabject “Class” may exist on UML class diagram and have its instances on UML object diagram. With such formalism an object can not exist without corresponding class, since it is its instance, and no special support is required to capture such relation in a visual modeling tool.

There are many publications about deep metamodeling (starting from [6] and including pivotal works [4], [5]) and there are some tools that use deep metamodeling as a metamodeling framework (for example, Melanee [7] and textual modeling tool MetaDepth [8]). But most of such tools are purely academic research projects or are supposed to be used by software engineering professionals, so whether deep metamodeling architecture is usable and beneficial for end-user modeling tools development, remains open question. Our research group had recently created rather successful visual programming tool employing DSM platform with two-level metamodeling architecture (in particular, QReal [9] DSM platform was used to create TRIK Studio [10] educational tool). A fact that resulting tool has several thousands of active users most of which are children who can not program in textual languages, is a good indication of feasibility of two-level metamodeling, but we experienced several language design problems that we were not able to solve with two-level architecture (see [10]). We decided to develop a new DSM tool using deep metamodeling architecture and to create an end-user programming tool using it to compare our experience with two-level metamodeling and to gain some experimental evidence on applicability of deep metamodeling to real-world end-user programming. "Smart greenhouse" programming tool was selected as our goal because it is relatively small and simple domain, but can be easily extended to more general Internet of Things applications, and there already exist some visual programming tools (Node-RED[1], for example) with which we can compare our results.

The remaining part of its article provides brief overview of deep metamodeling, followed by a brief overview of REAL.NET — our DSM platform that supports it. Then a visual language for "smart greenhouse" programming is introduced as an instance of REAL.NET metamodel hierarchy and tools for working with this language are described. Next we summarize and analyze our informal experience creating this tool, compare our results to related work and conclude the article. We believe that main contribution of this article is in reporting the experience of deep metamodeling usage for end-user programming tool development, which, we hope, can be meaningful contribution to empirical data related to metamodeling architectures, and as such helps to advance a knowledge about visual languages.

2. Deep Metamodeling

Deep metamodeling was first proposed in 2001 [4] as a basis of new UML 2.0 standard, but UML 2.0, released in 2005, still used two-level metamodeling architecture (and uses it by today). Later deep metamodeling received attention of domain-specific modeling research community, several tools using it were developed (i.e. [7], [8], [11]) and it sparked a wide interest and debates in related multi-level metamodeling techniques (see, for example, architecture based on "Powertype" pattern [12], comparison of different architectures in [13] and empirical study [14]).

Main idea of deep metamodeling is to consider entity in a model as a type and an object simultaneously. Each element of a model (node or edge) can be an instance of an element of some other model (which is then considered as metamodel) and a type for some elements in other models (so our model can be considered a metamodel related to them). Elements, attributes and various other elements have numeric attribute called "potency", which denotes how many times given element or attribute can be instantiated. For example, classic two-level metamodeling architecture can be considered deep metamodeling where potency of each element is either 1 or 0. Another example is a hierarchy of simplified UML metamodel, UML class diagram and UML object diagram, illustrated on figure 1.

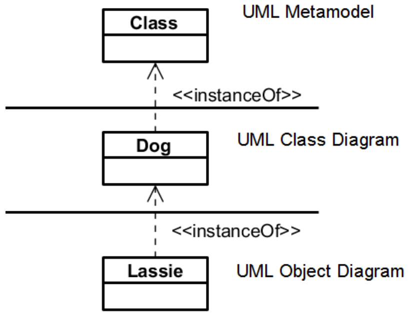


Figure 1. An example of class-object model hierarchy

Potency allows language creator to influence model structure several metalevels below language metamodel, which can be useful to connect user models with "instance-of" relations (like in our example with classes and objects). It is useful not only to explicitly express "instance-of", but also to give a limited ability for an user to specify language elements. It seems to be rather unexpected ability for a visual modeling tool, but has some important use cases — for example, user-created subprograms can be considered as a new language element, instances of "Subprogram", but able to have their own instances — subprogram calls.

For visual languages an editor shall be able to work with any element on a model, i.e. be able to correctly draw it, provide the ability to edit its attributes and so on. For this, all elements shall expose a set of properties that are not specified by their metamodel, but are determined by capabilities of an editor. To uniformly handle this, orthogonal metamodeling was proposed in [5] as an addition to "pure" deep metamodeling framework. With orthogonal metamodeling, each element is an instance of some element in a metamodel (which is called ontological metamodel) and at the same time it is an instance of some element in other — linguistic — metamodel, which determines only properties related to representation of an element. See figure 2 as an example of UML models hierarchy with an addition of linguistic metamodel.

3. REAL.NET Overview

REAL.NET is an implementation of DSM platform that is able to support deep metamodeling. We started working on REAL.NET with the aim to provide a tool and a set of libraries for experimenting with visual languages based on .NET platform, as an alternative to Eclipse Modeling Project, which is widely used for visual language research

1. Node-RED: http://nodered.org/. Accessed: 10.02.2019

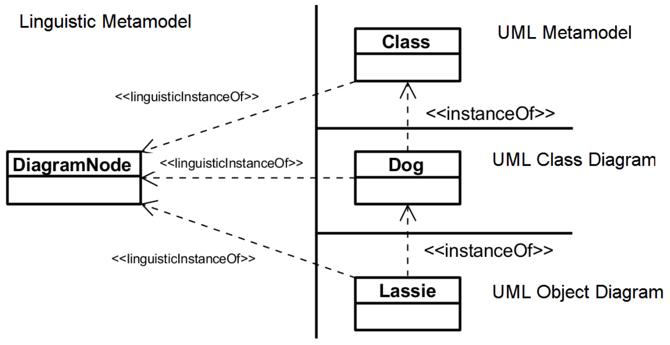


Figure 2. Orthogonal model hierarchy example

now. Eclipse Modeling Project targets Java platform and is highly dependent on Eclipse infrastructure, but there are many .NET applications that could benefit from visual languages, so we decided to develop our own DSM platform from scratch.

Component diagram with an overview of REAL.NET architecture is presented on figure 3.

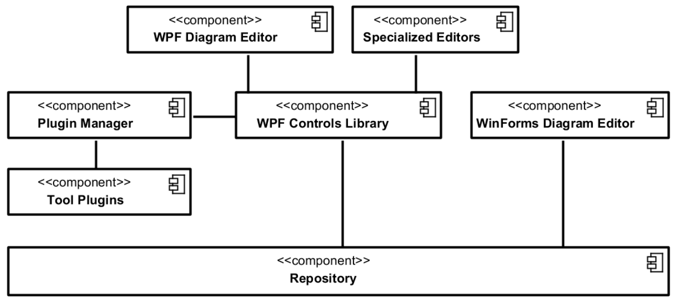

<<component>> WPF Diagram Editor
<<component>> Specialized Editors
<<component>> Plugin Manager
<<component>> WPF Controls Library
<<component>> WinForms Diagram Editor
<<component>> Tool Plugins
<<component>> Repository

Figure 3. REAL.NET high-level architecture

Main component of a system is repository. Repository is able to store models and perform operations on them, including operations that require knowledge of model semantics, for example, instantiation. Repository provides API for higher layers that allows them to work with models in high-level terms, for example, create elements that are instances of given type, having given properties. Repository encapsulates knowledge about metamodeling infrastructure and contains a hierarchy of predefined models which are needed to define that infrastructure and semantics. All data structures used by repository to store models are defined by those models themselves, so it is possible to automatically generate repository data structures using predefined models — a repository is self-defined in this sense.

Repository is used by two visual editors — one is based on WPF framework, other uses Windows Forms framework. WPF framework runs only on Windows but allows much nicer-looking GUI, Windows Forms framework, despite its name, runs nicely on Linux and Mac OS. Right now we consider WPF editor as our primary editor. WPF editor consists of reusable controls library which provides components like palette, scene, property editors and so on, and a frontend which puts these components together. We actually have several frontends — one for general-purpose diagram editing, others — for a specific programming tools based on REAL.NET, for example, quadcopter programming environment.

It is possible to construct domain-specific tool using components from controls library, but there is another possibility — define a plugin which will be dynamically loaded by plugin manager and is able to add its capabilities to an editor. Plugins are much simpler to implement than custom editor, so for scenarios where GUI is not important (e.g. code generators) plugins are preferred way to provide domain-specific functionality.

REAL.NET uses hierarchy of metamodels to define its metamodeling capabilities, and the "smart greenhouse" language is an instance of Infrastructure Metamodel, which is itself an instance of Language Metamodel, which is an instance of Core Metamodel, which is an instance of itself. This architecture allows us to relatively easily replace any metamodel layer and implement different metamodeling architectures.

4. REAL.NET Metamodels Hierarchy

The base metamodel in REAL.NET is Core Metamodel. The key element of metamodel is "Node" and it is the instance of itself, while the other elements that are defined in this model, such as "Element", "Edge", "Generalization", "Association" and "String" are the other instances of "Node" ("instance of" relation is itself modeled as "class" link from "Element" to itself). At the same time, all elements are inheritors of the "Element" and therefore should have an association relationship named "class" with some "element" inheritor. Full Core Metamodel is presented in figure 4, dashed lines represent "instance of" relations.

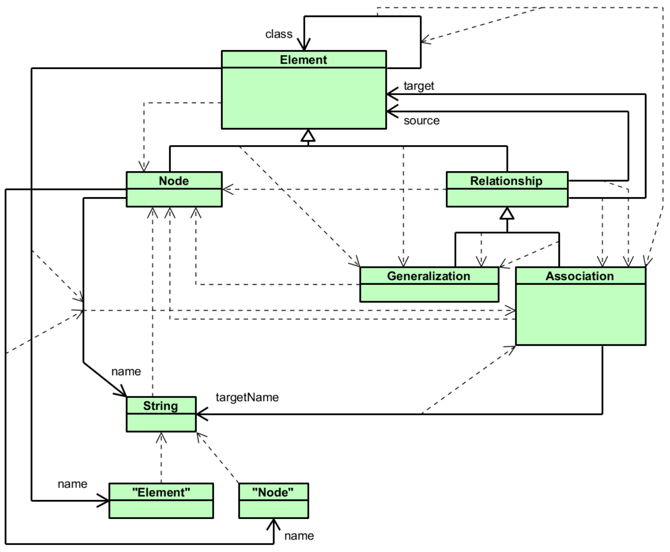


Figure 4. REAL.NET Core metamodel

The following level is the Language Metamodel. It defines the elements with which the Infrastructure Metamodel is built and contains edges, nodes and also definitions for "source"/"target" of edge and an "enum" type. The next level is Infrastructure Metamodel. It defines metamodeling capabilities that are used to create actual domain-specific languages, for example, this is the first model where "Attribute" notion is properly introduced. Infrastructure Metamodel models an interface between repository and editors and enables high-level operations such as creating a new model, creating elements and so on.

5. Visual Language

"Smart greenhouse" language is based on Infrastructure Metamodel and enables the creation of rules for autonomous greenhouse work using visual primitives. Consider two use case examples, when and to whom autonomous greenhouse work may be useful.

- Alice lives outside the city and has several greenhouses with different plant cultures. She spends a lot of time on opening/closing greenhouses every morning/evening when it gets warmer/colder. Therefore, she wants to automate this process, indicating in the scenario at what temperature windows should be opened in a specific greenhouse.
- Bob lives in the city, but has a greenhouse outside the city and has the opportunity to go there only on weekends. He wants to grow plants that require daily watering. Therefore, he needs the ability to set scenarios for watering plants depending on soil moisture.

Thus, the greenhouse scenarios should indicate which external conditions should trigger the device operation. Greenhouse program should react on the data that comes from sensors and send commands to actuators. So the paradigm of dataflow programming is useful to make the work more clear for end-users. Data flow is represented in form of edges that connect modules.

Greenhouse metamodel elements are shown on figure 5. There is only one association, it represents data flow in form of oriented edges that connect data sources with receivers. Source and target vertices should be descendants of the metamodel abstract node which is the instance of the infrastructure node. Since the entire system processes information from sensors and issues commands to the actuators, sensors are always sources and actuators are target nodes. The metamodel includes two types of sensors and three types of actuators. Each of them has port number of a physical device they introduce as an attribute. The metamodel abstract node also has logical operation and interval as descendants. In turn, the logical operation node has "AND" and "OR" operations as its descendants. The interval node has attributes of minimum and maximum values and sets an open interval for checking whether it has value that is passed to it. "Null" value of this attributes is interpreted as the absence of the upper/lower limit of the interval.

Each metamodel element also has the following attributes: a graphic figure defining its appearance when

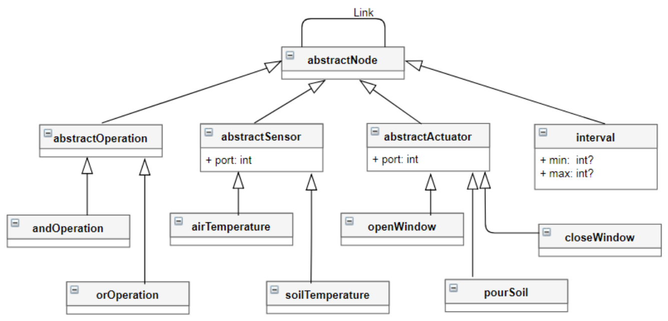


Figure 5. "Smart greenhouse" metamodel

visually building a script; "isAbstract" boolean type that indicates whether an element can be used in models; "instanceMetatype" — whether an instance of this element is edge or node.

The language is limited and, in particular, it is not possible to set specific attributes for each type of actuators. For example, the volume of watering for the device watering the soil.

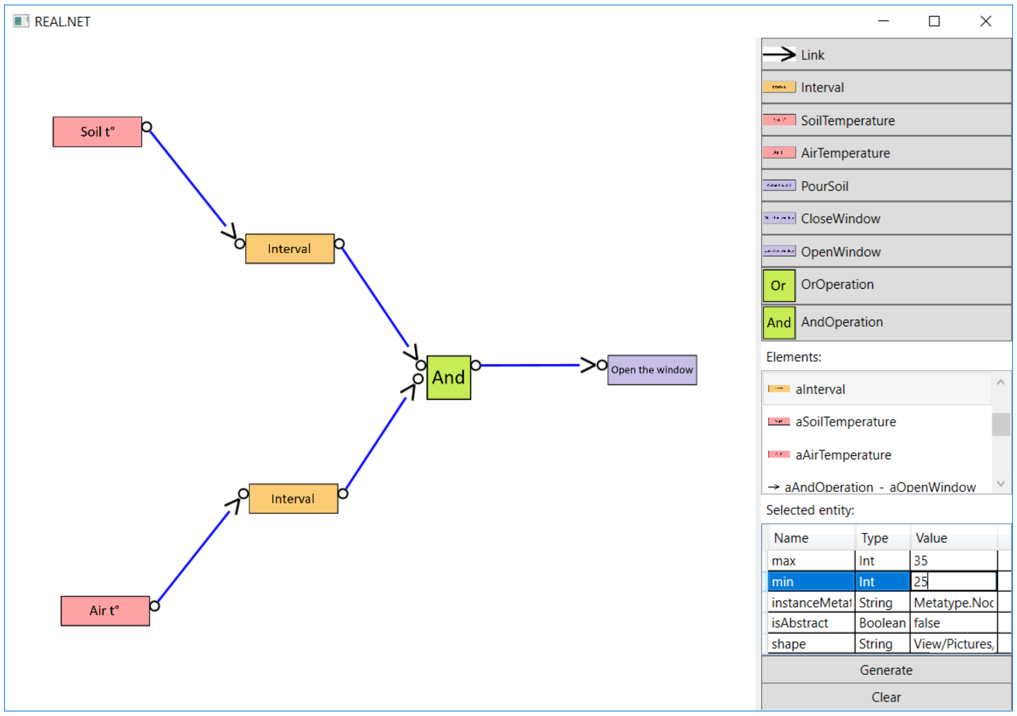
6. Tool Implementation

To model a scenario end-user drags all the necessary elements from the palette to the stage: actuators with sensors from his real greenhouse and operations with intervals for complex rules formation. Then he determines the values of all attributes of the selected nodes. And then he selects the edge in the palette and connects sources and receivers successively clicking on them. For the convenience of the user, the ability to draw ports on elements has been added to the editor — ports are the places on an element where edges can be connected. On the left side of the elements there are ports for incoming edges and on the right for outgoing. New free ports are automatically added to the node if all the others are already occupied but the element can still be linked to other vertices. User interface of our tool is presented om figure 6.

After the script is modeled, the user clicks on the "Generate" button on the right panel to generate code using the created model.

A generator has been implemented to produce an executable file that can be run on a micro-controller. It is written in C# and uses RX.NET and T4 technologies. T4 runtime text template takes model from repository as parameter and at the beginning creates an instance of the corresponding class for each element from this model assigning each an identifier. This identifier is used to set instance name and further to create dictionary with this identifier as a key where the "Operation" blocks can store values from all nodes connected to them. Property values of instances are set according to the attribute values from the model.

Our system should react and process the data that comes from different sensors. So each element should handle the



Name	Type	Value
max	Int	35
min	Int	25
instanceMeta	String	Metatype.Noc
isAbstract	Boolean	false
shape	String	View/Pictures,

Figure 6. "Smart greenhouse" prototype editor

events that the sources of its incoming vertices sends and then generate events for subsequent elements in the rule chain. To describe this behavior of the elements and to simplify work with event flows, the RX.NET library was used. Each node from the model is considered as the Subject in terms of Reactive Extensions. This means that it is at the same time the Observer, it reacts to changes in the preceding nodes, and the Observable, it allows to subscribe to its changes. Real greenhouse sensors and actuators or robot simulator sensors are considered to be Observables and Observers respectively.

Testing of the prototype required the use of a real micro-controller. We have used the TRIK micro-controller as it can run .NET virtual machine and real sensors and actuators can be easily connected to it. The Trik-Sharp library provides access to controller devices and even directly supports Reactive Extensions, representing sensors as Observables and actuators as Observers.

An example of the code generated by the model in figure 7 is shown in the listing 1. The "Air t" node is waiting for a value from the temperature sensor. The "Interval" node subscribed to "Air t" receives this value and checks whether it lies in the (5, 15) interval. And if it does, then the "Open the window" node receives TRUE value which means that it should send the open command to the actuator on the window.

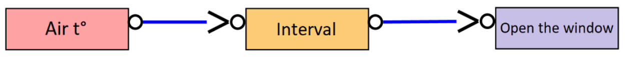


Figure 7. An example of a model

7. Experiment

To understand how easily end users can solve problems with the new language, a small test scenario was formulated.

"If the temperature sensor on port #1 shows a value greater than 20 degrees — open the window with an actuator on port #5. If less — close it with an actuator #4."

Four users were invited to model the scenario using the developed "smart greenhouse" solution. Two of them were not familiar with programming (the lawyer and the mechanical engineer). Two others were undergraduate students in software engineering familiar with UML but not with IoT. At the beginning of the experiment, work with the system was demonstrated on the example of similar scenarios. Users from the second group finished the work with the task faster, but on average it took only 3 minutes to model such scenario.

8. Lessons Learned

As mentioned earlier, "smart greenhouse" language metamodel includes only association relation and language itself is only a prototype of a full-featured programming

```
element0 = new Actuator(0);
element0.Port = 0;
IObservable<int> observable0 =
    System.Reactive.Linq.Observable.FromEventPattern<int>(
                h => element0.Event += h, h => element0.Event -= h)
                .Select(e => e.EventArgs).Synchronize().DistinctUntilChanged();
IObserver<int> observer0 = Observer.Create<int>(x => element0.Action(x));
ISubject<int> reactElement0 = Subject.Create<int>(observer0, observable0);

element1 = new Interval(1);
element1.Min = null;
element1.Max = null;
IObservable<int> observable1 =
    System.Reactive.Linq.Observable.FromEventPattern<int>(
                h => element1.Event += h, h => element1.Event -= h)
            .Select(e => e.EventArgs).Synchronize().DistinctUntilChanged();
IObserver<int> observer1 = Observer.Create<int>(x => element1.Action(x));
ISubject<int> reactElement1 = Subject.Create<int>(observer1, observable1);

element2 = new Sensor(2);
element2.Port = 0;
IObservable<int> observable2 =
    System.Reactive.Linq.Observable.FromEventPattern<int>(
            h => element2.Event += h,h => element2.Event -= h)
                .Select(e => e.EventArgs).Synchronize().DistinctUntilChanged();
IObserver<int> observer2 = Observer.Create<int>(x => element2.Action(x));
ISubject<int> reactElement2 = Subject.Create<int>(observer2, observable2);

var sub0 = reactElement1.Subscribe(reactElement0);
var sub1 = reactElement2.Subscribe(reactElement1);
```
Listing 1: Example of generated code

language for greenhouses. The set of language entities is clearly defined because it affects the code generation logic, which takes into account only specific sets of possible combinations. And types of sensors and actuators that can be installed in a real greenhouse are known in advance and taken into account when creating the language. Due to the fact that the end user does not need to invent additional classes when working with a greenhouse, during the work with this language there actually was no opportunity to take advantage of the deep metamodeling for end-user scenarios. From the language designer point of view, classic two-level metamodeling architecture was used — Infrastructure Metamodel and everything below it can be considered metametamodel, “smart greenhouse” language is metamodel and user scenarios of “smart greenhouse” control can be considered models. We believe that it is a typical situation in language design — to do something meaningful with models, custom code is needed, be it a generator, an interpreter, an analyzer and so on, and a need to write custom code greatly limits the flexibility of a metamodeling infrastructure.

On the other hand, deep metamodeling proved itself beneficial for DSM platform architecture. We implemented model persistence capabilities on Core Metamodel level, so each model derived from Core Metamodel can be stored/loaded. But editor capabilities are described by Infrastructure Metamodel, so we can have several different infrastructure metamodels for different visual language usage scenarios (for example, complex metamodel for an editor based on WPF and much simpler metamodel for a limited functionality editor based on Windows Forms). They are all can be based on the same Core Metamodel and take advantage of the functionality implemented in Core level. Our tool presented here did not require such capabilities, but we are planning to use them for web-based version, with completely different model editor requiring completely different infrastructure metamodel.

9. Related Work

There are some other systems that allow end users to create rules for devices sets. One of them is Node-RED[2] — a popular and widely used open source programming tool. It uses browser for scheme of interaction between devices definition. When compared with our system, the rules built in a similar way. The difference is the wide choice of blocks on the sidebar of Node-RED and some of them provide an opportunity to set complex rules by writing JavaScript functions. There are also many libraries that allow to expand the capabilities of the tool. But such an abundance of options for designing creates a sense of a complex system for beginners and have no sense in our greenhouse case, and a need for JavaScript programming makes it unsuitable for non-programmers.

There are many DSM platforms supporting deep metamodeling and, more general, multi-level metamodeling:

2. NodeRED, URL: https://nodered.org (accessed 10.02.2019)

Melanee [7], MetaDepth [8], WebDPF [11] to name a few. They are mainly created for research and concept-proof purposes and to our best knowledge there are no reports on their use for creation of actual domain-specific tools for end-users. Practical case studies like presented in this article are very limited. Research reported in [15] analyses features of 21 existing DSM tool with support of multi-level modeling, but lacks analysis of applicability of those features to real-world end-user applications. Work [16] considers methodological questions on evaluating multi-level modeling techniques and tools, but also is focused on technical qualities like model maintainability or model size, and does not report the experience of creating tools useful for non-programmers.

10. Conclusion

A new REAL.NET tool based on the deep metamodeling approach made it possible to quickly create a prototype of "smart greenhouse" graphical programming tool, which includes new language, editor and code generator. This prototype was tested with non-programmers and, as it seemed, turned out to be quite acceptable in real life for end users. Thus, in this article we reported a case study where a visual programming tool built using deep metamodeling architecture has enabled non-programmers to do limited programming tasks for a real-world scenarios.

Despite the fact that the area of "smart greenhouses" is rather narrow, it is part of the concept of the Internet of Things, and the result of this work can be extended to control sensors in any "smart" systems. All that is required is to determine the types of sensors that are possible to be installed in this area. And even if there are logical rules that are needed for processing values, but do not exist in the present system, the task of building a similar system based on this knowledge gained can be solved quickly using REAL.NET platform.

A logical continuation of the development of the application may be the transition to the web version. This will free the user from the obligation to install software on his own and also will help get rid of the need to create different versions for different platforms.

References

[1] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, p. 736, 2009.

[2] J.-P. Tolvanen and S. Kelly, "Metaedit+: defining and using integrated domain-specific modeling languages," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, pp. 819–820, 2009.

[3] J. Álvarez, A. Evans, and P. Sammut, "MML and the Metamodel Architecture," *WTUML: Workshop on Transformation in UML 2001*, p. 6, 2001.

[4] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel, "The Essence of Multilevel Metamodeling," *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, vol. 2185, no. JANUARY, pp. 134–148, 2001.

[5] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[6] JeanBézivin and R. Lemesle, "Ontology-based layered semantics for precise oa&d modeling," in *Proceedings of the ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*. Springer, pp. 151–154, 1997.

[7] C. Atkinson and R. Gerbig, "Flexible Deep Modeling with Melanee," *Modellierung 2016*, pp. 117–121, 2016.

[8] J. D. Lara and E. Guerra, "Deep Meta-modelling with MetaDepth," in *Objects, Models, Components, Patterns*. Springer, pp. 1–20, 2010.

[9] A. Kuzenkova, A. Deripaska, T. Bryksin, Y. Litvinov, and V. Polyakov, "QReal DSM platform-An Environment for Creation of Specific Visual IDEs," in *ENASE 2013 — Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*. Setubal, Portugal: SciTePress, pp. 205–211, 2013.

[10] D. Mordvinov, Y. Litvinov, and T. Bryksin, "TRIK Studio: Technical introduction," *2017 20th Conference of Open Innovations Association (FRUCT)*, pp. 296–308, 2017.

[11] F. Rabbi, Y. Lamo, I. C. Yu, and L. M. Kristensen, "WebDPF: A web-based metamodelling and model transformation environment," *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 87–98, 2016.

[12] C. Gonzalez-Perez and B. Henderson-Sellers, "A powertype-based metamodelling framework," *Software & Systems Modeling*, vol. 5, no. 1, pp. 72–90, 2006.

[13] C. Atkinson and T. Kühne, "In defence of deep modelling," *Information and Software Technology*, vol. 64, pp. 36–51, 2015.

[14] J. D. Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 12, 2014.

[15] M. Igamberdiev, G. Grossmann, and M. Stumptner, "A feature-based categorization of multi-level modeling approaches and tools," in *MULTI@MoDELS*. Ruzica Piskac, 2016, p. 11.

[16] C. Atkinson and T. Kühne, "On evaluating multi-level modeling," in *MODELS*, p. 4, 2017.