Towards a Systematic Engineering of Industrial Domain-Specific Languages

Rohit Gupta*, Sieglinde Kranz*, Nikolaus Regnat*, Bernhard Rumpe† and Andreas Wortmann‡

* Siemens AG, Munich, Germany

Email: rg.gupta@siemens.com, sieglinde.kranz@siemens.com, nikolaus.regnat@siemens.com

†Software Engineering, RWTH Aachen University, Aachen, Germany

Email: rumpe@se-rwth.de

‡Institute for Control Engineering of Machine Tools and Manufacturing Units, University of Stuttgart, Stuttgart, Germany

Email: andreas@wortmann.ac

Abstract—Domain-Specific Languages (DSLs) help practitioners in contributing solutions to challenges of specific domains. The efficient development of user-friendly DSLs suitable for industrial practitioners with little expertise in modelling still is challenging. For such practitioners, who often do not model on a daily basis, there is a need to foster reduction of repetitive modelling tasks and providing simplified visual representations of DSL parts. For industrial language engineers, there is no methodical support for providing such guidelines or documentation as part of reusable language modules. Previous research either addresses the reuse of languages or guidelines for modelling. For the efficient industrial deployment of DSLs, their combination is essential: the efficient engineering of DSLs from reusable modules that feature integrated documentation and guidelines for industrial practitioners. To solve these challenges, we propose a systematic approach for the industrial engineering of DSLs based on the concept of reusable DSL Building Blocks, which rests on several years of experience in the industrial engineering of DSLs and their deployment to various organizations. We investigated our approach via focus group methods consisting of five participants from industry and research qualitatively. Ultimately, DSL Building Blocks support industrial language engineers in developing better usable DSLs and industrial practitioners in more efficiently achieving their modelling.

Index Terms—Domain-Specific Languages, Model-Based Systems Engineering, Industrial Language Engineering

I. INTRODUCTION

There is a conceptual gap [1] in the systems engineering domain between the expertise of participating domain experts (biologists, chemists, mechanical engineers, etc.) and the challenges of systems engineering. Consequently, with the advance of various systems engineering domains from documents to models, we are seeing a shift in the way modelling is introduced at early stages of the systems engineering processes. The ubiquitous General-Purpose Languages (GPLs) used for software development present difficulties in system modelling [2] as they focus on technical implementation details, aggravate analysing the systems under development holistically, and prevent domain experts from contributing solutions directly. Domain-Specific Languages (DSLs) [3] instead aim to reduce the gap by being aimed at a particular domain, supporting domain-specific abstractions, and are better accessible to analysis and synthesis of systems and their parts. In the context of this paper, we use the term *developer* to refer to industrial language engineers and *user* to refer to industrial practitioners and domain experts.

Various graphical DSLs have been developed to support modelling in different domains, such as MATLAB Simulink [4] or SysML [5]. Yet, these are still overly generic and do not reflect domain concepts. However, systematically developing truly domain specific languages, e.g., a systems engineering language for the Italian railway system [6], that captures this particular domain's terminology (syntax), rules (well-formedness constraints), and meaning (semantics) is complicated. Reusing encapsulated DSL parts systematically can facilitate engineering new DSLs and ultimately foster truly domain-specific systems modelling. Additionally, the deployment of such DSLs to their users can be challenging. Often, the users the DSLs are developed for, model quite rarely, maybe once a week or less. Another challenge of graphical DSLs is to effectively represent DSL elements visually [7], [8] that improves usability heuristics and aids users in deriving hints to the meaning of such elements with the use of icons, colours and appearances. Hence, despite employing domain terminology, concepts, rules, and meaning, modelling with many DSLs can be less effective than expected. Industrial DSL engineering, therefore, needs to consider that users are perhaps modelling less often than expected and integrate modelling support and usability considerations into reusable language components, the DSL Building Blocks.

Addressing the above challenges is essential to achieving modelling goals effectively. Therefore, this paper presents an approach, DSL Building Blocks, which supports developers in building better usable graphical DSLs more efficiently and helps users achieve modelling more efficiently. Our approach, summarized in Fig. 1, separates the concerns of industrial engineering and deployment of DSLs along three different levels that relate to different skill sets and activities: (1) Concept level: in this level, developers define three parts: (i) the language, where abstract syntax, graphical concrete syntax and the translational semantic mapping is defined for developers; (ii) the method, where certain constraints and methodical steps are described for users to help achieve intended modelling goals; and (iii) the nucleus, where visual representations and notations on DSL elements are described by developers to help users relate to commonly used visual designs; (2) Tool-specific implementation level: in this level, developers realize the concepts described in the concept level by developing
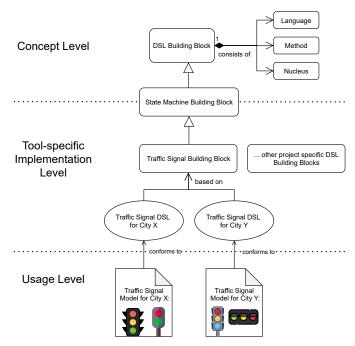
Fig. 1. A conceptual model for the development and usage of a graphical DSL describing the different levels in the development process including defining the DSL Building Block consisting of the method, language and nucleus at the concept level. Different traffic signals layouts for City X and City Y are depicted at the usage level for different cities.

the DSL Building Blocks and the DSLs using a graphical modelling tool; (3) Usage level: this is the level where users understand the methodical steps, documentation and visual representations of DSL elements to achieve their modelling goals. Leveraging this separation enables a systematic development of graphical DSL for developers who can reuse common parts for similarly structured DSLs. Further, it is beneficial to users as they can follow methodical steps and derive meaning from visual representation of DSL elements simplifying their modelling experience and efficiently reaching their modelling goals. Our focus on this paper is on the concept level, as developers could use different graphical modelling tools to build their DSLs. By defining DSL constraints, steps to reach a modelling goal, language syntaxes and nucleus nuances, we separate the language, method and nucleus of a DSL Building Block enabling developers to re-use these parts during the development of similarly structured graphical DSLs.

The concept of DSL Building Blocks is supported with a running example of a state machine and investigated through a case study and a qualitative focus group research [9]. Overall, our approach builds on years of experience in developing graphical DSLs for various industrial projects. By separating the concerns of industrial DSL engineering across different levels, developers achieve reuse of encapsulated parts in a systematic way. Effective visual representation of DSL elements and a structured guidance to achieving modelling goals is beneficial for users in optimizing their modelling experience.

In the remainder, II provides some background and related work, before III presents our approach with a running example. Afterward, IV describes a case study based on the example and V describes the evaluation. Finally, VI provides a discussion of the approach and VII concludes the paper.

II. BACKGROUND

DSLs are software languages and software languages are software too [10]. Hence, their engineering is subject to the usual challenges of software engineering in addition to considering multiple (meta)languages to define the language's constituents. Generally, a software language consists of [11], [12]: (1) an abstract syntax that defines the structure of its models, e.g., in form of grammars [13] or metamodels [14]; (2) a concrete syntax that defines how the models are presented, e.g., graphical [15], textual [16], or projectional [17]; and (3) semantics, in the sense of meaning [18], often realized through model-to-text [16] or model-to-model transformations [19].

To address this complexity, the research area of Software Language Engineering (SLE) [20], [21] has emerged and with it, language workbenches [22], specialized tools for the creation of software languages. Many of these provide advanced language engineering support, such as the generation of debuggers [15], editors [16], or reusable language modules [13] from abstract syntax descriptions.

Yet, methodologies for systematic SLE in the large are rare. And where studies detail how industrial graphical DSLs are implemented [23], [24], the reported methodologies are highly specific to individual departments and require repeated, time-consuming effort in language engineering. Either central research units, such as in Siemens Technology, must become more common in developing truly domain-specific or users in companies not having such units, such as Siemens Healthineers, must be trained with software and language engineering. Moreover, the language engineering and reuse methods proposed so far focus solely on technical improvements, such as explicit language interfaces [25], merging of language parts [15], or language types [26]. Usability of languages and language parts, in the form of modelling documentation, guidance, or automated modelling assistants as part of graphical DSL development are still missing.

In our literature review, we found two aspects that are often neglected: a systematic approach to developing graphical DSLs that fosters re-usability of common language elements combining guidelines for modelling and ways to achieve efficient user experience (UX) for users of such DSLs. Despite the efforts in previous research [27], [28], there is a lot to consider when building DSLs. We have not come across a methodology for developing graphical DSLs that takes into consideration how to reach a certain modelling goal and ways to improve the UX for users. One possible explanation could be that modelling goals vary for different projects and usability heuristics are not considered part of the language definition. In our experience of developing a diverse set of graphical DSLs for industrial projects, there is a need for a methodology that can be described independent of a specific implementation or graphical modelling tool that eventually benefits users.

III. DSL BUILDING BLOCKS

In this section, we introduce the approach and terminology of a DSL Building Block with the help of a running example

of a state machine. A state machine consists of *states* (the information of a system at an instant), *transitions* (translations from a source to a target state), and *triggers* (actions or inputs that hold to enable a transitions). *Initial* states describe where the state machine begins its execution or gets reset to. *Final* states describe where the state machine ends its execution. All other states are *intermediate* states.

A. DSL Building Block Structure

Our approach, summarized in Fig. 1 as a conceptual model, starts from a classical domain-driven approach where users define the business requirements such as number of states and transitions in a state machine. The systematic approach to developing graphical DSLs is based on domain-driven design [29], allowing developers to create graphical DSLs aimed at better modelling software and system architectures. These business requirements, specific to each project, are first translated into a *DSL Building Block* that primarily consists of three parts: (1) the *language*, which defines the abstract syntax, graphical concrete syntax and semantics of the language; (2) the *method*, which describes how to reach a modelling goal; and (3) the *nucleus*, which describes the visual representations and notations for better UX on model elements. In our example, defining these parts leads to a structured definition of a state machine DSL Building Block, which is then extended to specific industrial examples such as different traffic signalling systems, an oven or a heater, or systems that require sequential control logic. The outcome of this developmental approach is to prevent the reinvention of the same method, language and nucleus parts of similar graphical DSLs more than once.

The development and usage of a graphical DSL based on DSL Building Blocks is segregated into three logical levels and is performed by three actors whose tasks and activities are described in Fig. 2. At the *concept* level, a *DSL Building Block Developer*, who is a modelling expert with additional expertise in UX, and language engineering (such as key expert engineers in software and systems research units at Siemens), describes the constraints and method of use of the DSL, the abstract syntax, the graphical concrete syntax along with a structured documentation of model elements and the nucleus consisting of visual notations and representations of the model elements in accordance with the requirements specified by users. As well as defining the parts generically, such as for a state machine, they also identify and define project specific DSL Building Block requirements, such as for a traffic signal system. The *tool-specific implementation* level is where a *DSL Developer*, equipped with sufficient programming skills needed for building graphical DSLs, selects the relevant graphical modelling tool in accordance to their organization and develops the DSL Building Block and its corresponding DSL. The *usage* level is where users understand the steps, documentation and visual representations of DSL elements to ultimately achieve their modelling goals on the selected graphical modelling tool. This is performed by a *DSL User*, who does not need to be proficient in UX, modelling or programming, but possess expert knowledge in their respective domains. They also provide subsequent feedback and improved requirements

to DSL Building Block Developers corresponding to agile principles of immediate feedback and continuous integration [30]. As DSL Users represent a diverse set of users, it is important for a DSL Building Block Developer to describe the parts of a DSL Building Block in a meaningful and easy to understand manner using appropriate documentation and visual designs and notations belonging to a particular domain. While developers could also possess domain expertise, they are rather uncommon in Siemens.
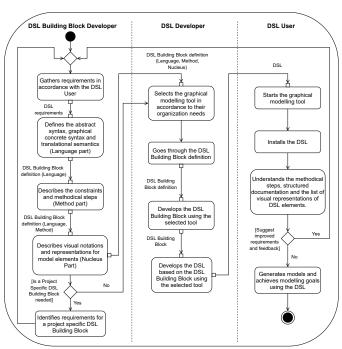

Fig. 2. An activity diagram describing the tasks and activities of the three actors in the DSL Building Block approach.

We now describe each of the parts based on the state machine example. It is worth mentioning that we represent the state machine model elements within square brackets such as [State: X], [Transition: y]. X and y are the names or an identification number for the respective model element in a derived DSL Building Block. For a state machine DSL Building Block the main model elements are defined as [State], [Transition] and [Trigger].

B. The Language

The *language* part of the DSL Building Block describes the abstract syntax, the graphical concrete syntax and the translational semantic mapping for the language. Fig. 3 shows the abstract syntax of a state machine describing the necessary concepts and structure of the language. The model elements [State Machine], [State], [Transition], [Trigger] and the [StateType] are defined along with the list of attributes. In our example, the method list is left empty for the class diagram, but can be modified later to allow for specific business requirements. The DSL Building Block Developer initially defines a default graphical form without any specific visualization properties, for each model element as part of the concrete syntax. E.g., a state is defined to be a rectangle corresponding to a UML object and a transition is defined to be

a straight line. These model elements are visually defined and improved later in the nucleus to complete a better graphical concrete syntax aimed for a more visually understandable and easy to use DSL. The structured documentation from the defined model elements to the semantic domain essentially provides meanings to those model elements. For the state machine, a structured documentation of the model elements and their attributes along with detailed description is shown in Table I, for DSL Users to easily understand and relate each model element.
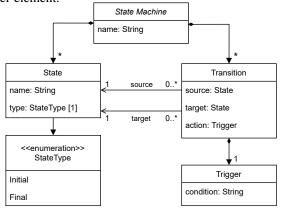

Fig. 3. A class diagram showing the abstract syntax of a state machine DSL Building Block as well as the relationships between them.

C. The Method

The *method* part of the DSL Building Block describes the modelling goal for a business use case, which serves as a guide to a DSL User, and addresses the question "how-to" reach that goal. Modelling goals vary for DSL Users and include representations of target systems and their behaviour, ideas, simulations as well as specific business requirements for a wide variety of projects. An example of a business use-case would be to design a traffic signal system that follows a sequential control logic. Given the variety of domains and projects, there is a need for a comprehensive guide for DSL Users to reach their goals in an effortless manner. The DSL Building Block Developer describes a suitable approach to reach these modelling goals using a sequence of methodical steps, specified textually and using an activity diagram. The inputs and outputs at each step consists of an actual model, parts of a model or trivial and non-trivial business requirements in relation to the composition of the system in consideration. In addition, a list of constraints is defined for the DSL Developer to ensure that the basic, yet critical, conditions for a DSL Building Block, that may otherwise be overlooked, are pre-checked and validated. The methodical steps provide a helping guide to DSL Users in reaching their modelling goals.

The list of constraints for a finite state machine, which consists of a finite number of [State]s and [Transition]s, are described as follows by the DSL Building Block Developer. *Constraint 1:* All the [State]s described in the State Machine DSL Building Block must be reachable by [Transition]s, except the initial [State] which may or may not be reachable by a [Transition] but is the starting point of the machine. *Constraint 2:* A [State] can have more than one incoming [Transition]s,

each from different [State]s. Similarly, a [State] can have more than one outgoing [Transition]s, each to different [State]s. *Constraint 3:* The initial [State] has either zero or one incoming [Transition] from an intermediate [State] and the final [State]s will not have any outgoing [Transition]s. *Constraint 4:* A [Transition] between two [State]s must execute within a defined time frame.

Fig. 4 describes the steps needed to reach a modelling goal for a state machine making it beneficial for a DSL User to easily understand and follow the process needed to reach their modelling goals.
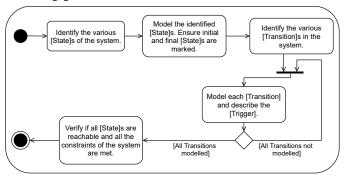

Fig. 4. A method activity diagram that describes the sequence of methodical steps needed to reach the modelling goal for the state machine DSL Building Block example.

D. The Nucleus

The *nucleus* part of the DSL Building Block consists of various characteristics of model elements for the language, termed *nucleus nuances*, that help provide a coherent UX to users based on context conditions, visual representations, transformations and validation rules. Context conditions are boolean predicates on a language's abstract syntax to checking its consistency and is used to determine if a model is well-formed [13]. Visual representations are representations of model elements in the form of icons, colours, appearance, dialogs and its properties in relation to shape, size and opacity. Transformations allow model elements to be automatically instantiated and validation rules enable better error detection with model elements, including checking redundant model elements or misconfigured types that are hard to be detected manually. Certain studies have explored generating graphical syntax for better visual representation [31], [32]. A *nucleus nuance* is a characteristic of a model element describing the model element's intent, motivation and consequences based on reasoning to enhance UX for DSL Users. The usability aspects in building graphical DSLs is often neglected which leads to users struggling in understanding complex DSLs [33], [34]. Therefore, nuances are described by the DSL Building Block Developer for effective usability and visual notation conventions [35], [36], [7] in making better design decisions with respect to each individual domain. Each nuance is described with a reasoning as a textual template, that the DSL Developer builds into the DSL Building Block and the DSL. Users can thus easily understand the notations and importance of these nuances for effectively using a graphical DSL. We now list a few nucleus nuances for the state machine example.

Syntax Element	Attribute	Description
State Machine		A finite state machine with a fixed number of [State]s and [Transition]s.
State		Representation of information of a system at a given point.
	name	Name of the [State].
	type	Type of a [State]: *Initial*, *Intermediate*, *Final*.
Transition		A path between two [State]s based on an action.
	source	A [Transition] starts at this [State].
	target	The [State] where the [Transition] ends.
	action	The [Trigger] that switches the [Transition] from a source to a target [State].
Trigger		A logical condition for a [Transition] running for a definite period of time.
	condition	A string holding the condition requirement.

Nuance 1: On creation of a [State Machine], an initial [State] is also created automatically on a graphical modelling tool canvas. *Reason*: Users often forget to create or mark the initial state or remove it without marking another initial state during model creation or refactoring.

Nuance 2: [State]s are oval or circular in shape and [Transition]s are denoted with curved black arrows. *Reason*: Representing different model elements in particular shapes allows for easy visual identification of model elements on the graphical modelling tool.

Nuance 3: The initial [State] is marked with an **i** (alphabet i) symbol whereas the final [State]s are marked with an **f** (alphabet f) symbol. *Reason*: Visualizations with different symbols prevent users from creating multiple such [State]s or confuse them with other [State]s in a complex system with multiple model elements.

Nuance 4: All instances of a [State] are filled with a distinct colour, except intermediate [State]s, where two such intermediate [State]s can be filled with the same colour. *Reason*: Visualizations with different colours help distinguish [StateType]s.

Nuance 5: A [State] without an incoming or outgoing [Transition]s is marked with a red exclamation mark **!** at its top right corner. *Reason*: Often in complex models, changes in the model leads to the unwanted removal of model elements leading to errors in model. This nuance, thus, helps in validation and error detection.

Nuance 6: A [Transition] between two [State]s is represented by a curved black line. If a [Transition] does not contain a [Trigger], the link is coloured red along with an exclamation mark **!**. *Reason*: This nuance also helps in detecting errors and validates certain rules that may be thought of initially as an implied behaviour.

Nuance 7: Each [State]s can also be marked with any additional relevant icon that represents visually aiding information about the [State]. *Reason*: Complex industrial systems consists of multiple hardware and software resources. Using relevant icons helps identify model elements with ease.

IV. EXTENDED EXAMPLE AS A CASE STUDY

A local government of a city is holding an exhibition (expo) inviting industrial manufacturers to foster innovation around mobility and sustainability. To effectively manage traffic during the event, an intelligent traffic management system is needed serving different purposes. For example, one area of the expo calls for a traffic signal serving pedestrians and cars, while another area of the expo needs a traffic signal catering to fully autonomous cars, thus requiring fewer states and transitions and standardised traffic signal lights that the autonomous cars can detect to either proceed or stop. With a state machine DSL Building Block defined earlier, deriving such traffic signal DSL Building Blocks, that also require sequential control logic, becomes significant for re-usability. This reduces repetitive tasks such as defining constraints, states and transitions and defining syntax related to the language during the graphical DSL development.
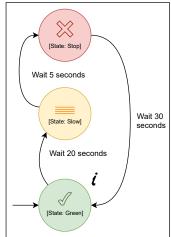
We extend the example of a state machine to a traffic signal in this case study. A traffic signal DSL Building Block consists of three state instances: [State: Go] (Initial State), [State: Slow] (Intermediate State), [State: Stop] (Intermediate State). The transitions and triggers for these states are shown in Table II. The times in the trigger column are filled in by DSL Users when building models, thus simplifying the use of a DSL. The method, language and nucleus parts are adapted by the DSL Building Block Developers for a specific traffic signal and do not influence the state machine DSL Building Block. However, an advantage of exploring different case studies allows common adaptations to also be incorporated directly in a parent DSL Building Block for future re-use, thus allowing quick and continuous improvements. An example of such commonality could be a sensor that is used in all instances of all traffic signal implementation. Finally, nuances help in relating to better visual representations allowing DSL Users to design traffic signal models that are similar to real-world traffic signals as shown Fig. 5.

We now discuss modifications to the state machine example to support the traffic signal DSL Building Block. The following constraint is added to the method. *Constraint 5:* As a traffic signal is intended to run indefinitely, the final state will not exist. The methodical steps from the state machine example is updated to include the following transitions. [Transition: 1]: [State: Go] ->[State: Slow]; [Transition: 2]: [State: Slow] -

Transition	Source State	Target State	Trigger Condition
[Transition: 1]	[State: Go]	[State: Slow]	"Wait t_1 seconds"[a]
[Transition: 2]	[State: Slow]	[State: Stop]	"Wait t_2 seconds"[a]
[Transition: 3]	[State: Stop]	[State: Go]	"Wait t_3 seconds"[a]

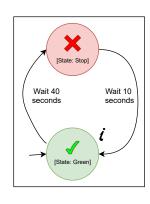[a]t_1, t_2, t_3 are the times defined by a DSL user when using the DSL.

Fig. 5. A traffic signal model for pedestrians and cars (left) and a model for autonomous cars with fewer states and transitions (right).

>[State: Stop]; [Transition: 3]: [State: Stop] ->[State: Go]. The class diagram representing the abstract syntax is updated to reflect traffic signal specifics, such as two state types: Initial and Intermediate, and three states. The documentation of the model elements described in Table I is updated to reflect the changes. Transitions and triggers for this traffic signal example are described in the Table II. To further improve visual aspects for the traffic signal, the following nuances are updated or added. *Nuance 2 (updated):* The states must be circular in shape and are arranged vertically in this order (top to bottom): [State: Stop], [State: Slow] and [State: Go]. *Nuance 8:* [State: Go] is filled with green colour and contains a green tick (✔) icon. *Nuance 9:* [State: Slow] is filled with yellow colour and contains an orange (≡) icon. Finally, *Nuance 10:* [State: Stop] is filled with red colour and contains a red cross (✖) icon. The reasoning for these nuances is to effectively represent important model elements visually by defining colours, shapes and layouts that are commonly used in real world traffic signals. This case study shows methodical steps, structured documentation and better visual notations of model elements that help DSL Users model systems with a greater degree of confidence and show how parts of a similarly structured graphical DSL can be re-used with minimal adaptations.

V. EVALUATION

The proposed approach was subject to a qualitative assessment using focus group methods [9]. The purpose of this evaluation was to bring together a group of experienced practitioners and researchers to collectively understand the challenges of the approach, discuss possible solutions and define a systematic graphical DSL developmental approach that would

be beneficial for both practitioners and researchers alike. In this section, we describe the evaluation pre-processing, discussions and the results. The evaluation was initially planned for two in-person phases, however due to travel and contact restrictions surrounding COVID-19, the second phase was conducted online using video conferencing tools.

A. Participants

To assess the approach, five participants with varying modelling and programming knowledge were chosen. Two participants from the industry were domain modelling experts with 8-12 years experience in developing graphical DSLs. Two participants were researchers from the software engineering domain with limited programming, but 6-12 years experience in software and systems modelling. The final participant was a software developer with limited modelling experience, but 5 years of programming and UX skills. The moderator, a research group manager with experience as a scrum master, was briefed with the proposed approach before each phase.

B. Phase 1

Phase 1 was held in-person. The focus group was guided by the moderator who prepared a list of questions and activities on the proposed approach with the discussion lasting two hours.

Pre-processing. The moderator was given three weeks to prepare the content of the discussion, the scripts and the technical setup. The outcome of this phase was to provide a first look into the problems faced by domain experts in building graphical DSLs for industrial projects, and to foster lively discussions and feedback on the proposed approach. The moderator proposed the prepared set of questions and activities, including whiteboard discussions, and took handwritten notes while guiding the discussion.

Discussion and results. The discussion was held over three stages: introduction, main stage and the follow-up phase. In the introduction stage, the moderator introduced the participants and presented the elementary question: (1) *What is the single most difficult challenge when designing DSLs?* The practitioners and researchers unanimously agreed on the need to have proper guidelines in developing graphical DSLs. In the main stage, the following questions were asked: (2) *Which steps in designing graphical DSLs consumes the most of your time?* (3) *Is the usage of visual notations beneficial when using a graphical DSL?* In this stage, there were mixed opinions. While practitioners favoured UX as being more beneficial, the researchers said re-usability helps in reducing time-consuming tasks. Then, the difference between the language and nucleus parts was discussed as certain nuances could be part of the syntax of the language. However, one researcher opined that segregating the nuances and classifying them with reasoning would be more structured for developers. The software developer advocated the use of common visual notations and representations making DSL elements easier for any user to understand. In the follow-up phase, all participants believed more clarifications were needed for phase two as some issues of this approach were insufficiently addressed.

C. Phase 2

Phase 2, initially planned for in-person, was conducted online, three months later, due to restrictions surrounding

COVID-19 with the same participants and moderator. Building on the experience in the first phase and being updated with an improved version of the approach, the moderator framed a diverse set of questions and activities for this phase.

Pre-processing. The moderator prepared the content of the discussion and set up the audio and video conferencing in Microsoft Teams [37]. The whiteboard discussion was held over Conceptboard [38]. The outcome of this phase was to reach a systematic approach beneficial for both users and developers of graphical DSLs.

Discussion and results. The discussion was held over three stages: introduction, main stage and the follow-up phase. In the introduction, the moderator presented the approach and the initial question: (1) *What strengths or weaknesses does this improved approach carry?* This was followed by a creative discussion in the main stage which included virtual whiteboard discussions. In this stage, participants came up with an approach to define various constraints in the method part and reasoning for each nuance based on the state machine example. At times the audio and video distorted, however, the moderator noted that there was no real disruption in the outcome as all topics were well articulated. The inability to use an offline whiteboard was a challenge for participants as more time was needed here than planned. In the follow-up phase, participants were asked to describe and rate the further improved approach and list prospective work. The participants agreed this final systematic approach is beneficial for both developers and users as well as practitioners and researchers. The final approach presented in this paper is a three layered developmental approach incorporating various suggestions and feedback based on both focus group discussions.

VI. DISCUSSION

The presented methodology enables DSL Developers to systematically develop graphical DSLs. We achieve re-usability of common aspects during graphical DSL development by separating the concerns of industrial DSL engineering along three different levels relating to different skill sets and activities. The approach, DSL Building Blocks, assists DSL Building Block Developers in defining constraints and methodical steps intended to help DSL Users achieve their modelling goals. The business requirements for each project specific DSL is gathered by DSL Building Block Developers in consultation with DSL Users. The developer then extracts commonalities from requirements in the form of a DSL Building Block definition. Project specific use-cases of DSL Building Blocks can extend and adapt the method, language and nucleus parts, thereby providing adequate flexibility to domain experts in realizing problems of a specific domain. Furthermore, visual representation of various DSL elements is beneficial for DSL Users in simplifying their DSL modelling experience, as users can relate such elements to real world domain specific examples. The combination of providing methodical steps to reach a modelling goal and the focus on UX for DSL Users has been addressed with our approach. Our approach presents a structured graphical DSL development process including adaptation and continuous integration of DSL requirements

from DSL Users, who closely interact and suggest feedback to DSL Building Block Developers. Our approach to DSL engineering is limited to graphical DSLs, as large corporations such as Siemens, mostly focus on visualization concepts for representing DSLs. We note that graphical modelling tools such as MetaEdit+ [39] and MagicDraw [40] have different technical capabilities, which therefore poses a challenge towards adopting this methodology seamlessly across all tools. Other language engineering tools such as MPS [41], Spoofax [42], and Melange [15] provide certain means for language composition and customization, but fail to provide methods for systematic reuse for similarly structured DSLs. While our focus group evaluation was limited to a few practitioners and researchers, we are currently building an extensive survey to gather ways on improving UX for domain experts within Siemens. As part of the ongoing research, we plan to categorize and structure various nuances allowing for them to be easily analyzed to make it more machine-processable and accessible to automation. Further, we plan to work on challenges to introduce inheritance support for multiple DSL Building Blocks in enabling re-use support across a wider variety of domains not structurally related to each other.

Overall, our approach builds upon our experience of developing graphical DSLs for various industry projects as well as the qualitative evaluation of the approach using focus groups that included experienced practitioners and researchers. In this vision, the need to have a proper set of guidelines and documentation is important. We, therefore, propose this systematic approach that not only fosters re-use of language parts for DSL Developers, but also focuses on visual representation of model elements that help DSL Users understand and use graphical DSLs with simplicity. We are unaware of any other similar methodologies for the development of graphical DSLs that helps realize this vision.

VII. CONCLUSIONS

We have presented a systematic approach for developing graphical DSLs, DSL Building Blocks, through separation of concerns of industrial DSL engineering among developers and users with different skill sets and activities. This approach solves challenges related to re-usability of common DSL elements for developers and is intended for a better user experience for domain experts who possess different skill sets in DSL language use and engineering. While our approach is currently limited to graphical DSLs, they greatly facilitate the re-use of DSL parts and provide robust guidance, documentation and effective visual representations to users helping them achieve modelling goals with simplicity. Continuous feedback by users to developers help in constantly adapting and improving the DSL Building Blocks and subsequent DSLs. This fosters the adoption of the systematic development of graphical DSLs among developers and narrows the gap between practitioners and researchers. As part of our ongoing research, we are building an extensive feedback survey to collect experiences from users in improving our methodology. We also plan to apply our approach to different domains by

introducing inheritance support between DSL Building Blocks and integrate further language definition dimensions.

REFERENCES

[1] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," *Future of Software Engineering (FOSE '07)*, pp. 37–54, May 2007.

[2] H. A. Proper and M. Bjeković, "Fundamental challenges in systems modelling," in *40 Years EMISA 2019*, H. C. Mayr, S. Rinderle-Ma, and S. Strecker, Eds. Bonn: Gesellschaft für Informatik e.V., 2020, pp. 13–28.

[3] M. Fowler, *Domain-Specific Languages*, ser. The Addison-Wesley signature series. Addison-Wesley, 2011.

[4] S. J. Chapman, *MATLAB programming for engineers*. Nelson Education, 2015.

[5] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2014.

[6] B. Meyers, M. Wimmer, A. Cicchetti, and J. Sprinkle, "A generic in-place transformation-based approach to structured model co-evolution," *Electronic Communications of the EASST*, vol. 42, Jan. 2011.

[7] D. Moody, "The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, p. 756–779, Nov. 2009.

[8] J. Nielsen, "Designing web usability," 2000.

[9] R. Krueger and M. Casey, *Focus Groups: A Practical Guide for Applied Research*. SAGE Publications, 2009.

[10] J.-M. Favre, D. Gasevic, R. Lämmel, and E. Pek, "Empirical language analysis in software linguistics," in *International Conference on Software Language Engineering*. Springer, 2010, pp. 316–326.

[11] M. V. Cengarle, H. Grönniger, and B. Rumpe, "Variability within Modeling Language Definitions," in *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, ser. LNCS 5795. Springer, 2009, pp. 670–684.

[12] T. Clark, M. v. d. Brand, B. Combemale, and B. Rumpe, "Conceptual Model of the Globalization for Domain-Specific Languages," in *Globalizing Domain-Specific Languages*, ser. LNCS 9400. Springer, 2015, pp. 7–20.

[13] K. Hölldobler and B. Rumpe, *MontiCore 5 Language Workbench Edition 2017*, ser. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[14] B. Combemale, O. Barais, and A. Wortmann, "Language engineering with the GEMOC studio," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 189–191.

[15] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A Meta-language for Modular and Reusable Development of DSLs," in *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, 2015.

[16] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[17] F. Campagne, *The MPS language workbench: volume I*. Fabien Campagne, 2014, vol. 1.

[18] D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of "Semantics"?" *IEEE Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[19] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of computer programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[20] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education, 2008.

[21] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software Language Engineering in the Large: Towards Composing and Deriving Languages," *Computer Languages, Systems & Structures*, vol. 54, pp. 386–405, 2018.

[22] S. Erdweg, T. Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, "The State of the Art in Language Workbenches," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, M. Erwig, R. F. Paige, and E. Van Wyk, Eds. Springer International Publishing, 2013, vol. 8225, pp. 197–217.

[23] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry, "Leveraging software product lines engineering in the development of external dsls: A systematic literature review," *Comput. Lang. Syst. Struct.*, vol. 46, pp. 206–235, 2016.

[24] J. Tolvanen and S. Kelly, "Defining domain-specific modeling languages to automate product derivation: Collected experiences," in *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, 2005, pp. 198–209.

[25] A. Butting, J. Pfeiffer, B. Rumpe, and A. Wortmann, "A Compositional Framework for Systematic Modeling Language Reuse," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, October 2020, p. 35–46.

[26] J. Steel and J.-M. Jézéquel, "On model typing," *Software & Systems Modeling*, vol. 6, no. 4, pp. 401–413, 2007.

[27] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design Guidelines for Domain Specific Languages," in *Domain-Specific Modeling Workshop (DSM'09)*, ser. Techreport B-108. Helsinki School of Economics, October 2009, pp. 7–13.

[28] G. Czech, M. Moser, and J. Pichler, "A systematic mapping study on best practices for domain-specific modeling," *Softw. Qual. J.*, vol. 28, no. 2, pp. 663–692, 2020.

[29] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[30] K. L. Beck, *Extreme programming explained - embrace change*. Addison-Wesley, 1990.

[31] S. Maro, J. Steghöfer, A. Anjorin, M. Tichy, and L. Gelin, "On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, 2015, pp. 1–12.

[32] B. Nastov and F. Pfister, "Experimentation of a graphical concrete syntax generator for domain specific modeling languages," in *Actes du XXXIIème Congrès INFORSID, Lyon, France, 20-23 Mai 2014*, 2014, pp. 197–213.

[33] E. Mosqueira-Rey and D. Alonso-Ríos, "Usability heuristics for domain-specific languages (dsls)," in *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, 2020, pp. 1340–1343.

[34] I. P. Rodrigues, A. F. Zorzo, M. Bernardino, and M. de Borba Campos, "Usa-dsl: usability evaluation framework for domain-specific languages," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pp. 2013–2021.

[35] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, 1996.

[36] A. F. Blackwell, "Cognitive dimensions of notations: Understanding the ergonomics of diagram use," in *Diagrammatic Representation and Inference, 5th International Conference, Diagrams 2008, Herrsching, Germany, September 19-21, 2008. Proceedings*, 2008, pp. 5–8.

[37] (2020) A collaborative workspace offering workspace chat and video conferencing. [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/microsoft-teams/group-chat-software

[38] (2020) An online visual collaboration workspace. [Online]. Available: https://conceptboard.com/

[39] J. Tolvanen, "Metaedit+: integrated modeling and metamodeling environment for domain-specific languages," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, 2006, pp. 690–691.

[40] D. Neuendorf, "Review of magicdraw uml® 11.5 professional edition," *J. Object Technol.*, vol. 5, no. 7, pp. 115–118, 2006.

[41] M. Völter and E. Visser, "Language extension and composition with language workbenches," in *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, 2010, pp. 301–304.

[42] G. H. Wachsmuth, G. D. Konat, and E. Visser, "Language design with the spoofax language workbench," *IEEE Software*, vol. 31, no. 5, pp. 35–43, 2014.