EASST

7th International Symposium
on Leveraging Applications of Formal Methods, Verification and Validation

-

Doctoral Symposium, 2016

Towards Reuse on the Meta-Level

Dominic Wirkner, Steve Boßelmann

11 pages

Towards Reuse on the Meta-Level

Dominic Wirkner[1]**, Steve Boßelmann**[2]

[1] dominic.wirkner@udo.edu
[2] steve.bosselmann@cs.tu-dortmund.de

Chair for Programming Systems, Department of Computer Science, TU Dortmund University,
44227 Dortmund, Germany

Abstract: Modern software development peaks in complex product lines and utilizes features of programming languages to their full extend. On the other hand, model driven development shines by abstraction from implementation details to ease communication between programmers and domain experts. In this paper an extension to the language family of the CINCO Meta Tooling Suite is proposed, which allows a more flexible and efficient way to reference elements between different models. The definition of how representational elements reference their origins in another model is shifted away from code-level to make it accessible to domain experts. It also enables the possibility to reuse such specification without the need for code or model duplication. In addition, validation and needed updates between referenced elements and their representation is supported by a centralized transformation definition. This highly stretches reuse of implementation to overcome reoccurring problems in the context of inter-model references and acts as one part of a foundation for developing software product lines with the CINCO framework.

Keywords: model driven development, software product lines, cinco meta tooling suite, graphical modeling

1 Research Motivation

Since the early days of programming developers tried to optimize development time and maintenance costs to ensure a desired level of quality. One key part in this challenge is minimizing code duplication. This is getting more and more difficult due to increase of complexity in modern software systems. Nowadays this complexity is reflected by the emerging use of Software Product Lines (SPLs). Classical SPLs consist of a core part and a collection of features, which some of them are mutually exclusive and other ones are purely optional. The definition of rules for feature combination builds its own scientific field [BPS04, LNS13, VV11] and achieving a desired level of source code reuse requires an implementation language to have sufficient expressive power.

Typical continuous evolution of SPLs, which is supported by version control systems, faces the problems of software variants. This leads to a variability problem in space and time [SSA14]. Finding a language that is capable of dealing with different variants and versions of software components, which means definition (design-time) and execution (runtime) of behavior, is another challenge to solve. General-purpose programming languages are capable to offer a solution for this problem, but they disadvantage application experts that typically lack programming knowledge.

Through history, a major motivation in the design of programming languages has been the strive for simplicity in terms of language features that enable definitions in a simple yet precise manner. To achieve simplification for a specific domain the introduction of new language features might even come along with the sacrifice of expressiveness. As an example, the replacement of GOTO statements by means of various loop constructs means a reduction in expressive power. On the other hand, this step made it possible to define *what* happens without the need to dictate *how* it is performed. Such a shift from *how* to *what* has repeatedly been motivation for the development of domain-specific languages that pursue a declarative approach instead of using imperative instructions.

Within the last decade the paradigm of Model Driven Development (MDD) has become quite popular because of its ability to narrow the gap between application experts and software developers [SM09]. Furthermore, providing a One-Thing Approach (OTA) i.e. a common and corresponding software is key for reaching such goal [MS09]. Driven by the aspect of simplicity [NNL+13] and full code generation [KT08] the CINCO Meta Tooling Suite has been developed [NLKS17], which supports an application expert by generating a graphical model editor from defined meta models. The analysis of the development process based on various applications of CINCO [BNNS16, BFK+16] highlighted possibilities for improvements in reusing parts of meta models to increase the benefits of OTA even further.

Up to this point, general-purpose languages are capable of defining SPLs, but application experts that rely on domain knowledge instead of source code can hardly access them. On the other hand, modeling environments like those created with CINCO support application experts, but lack the possibility to implement SPLs. There are some first concepts to enhance CINCO by bootstrapping new features and reusing implemented effort [NNMS16]. However, achieving a complete model driven SPL development with CINCO requires enhancements on the core language concepts.

One major problem to overcome is a specification of meta model families and the linking between their languages without redefinition and duplication. The first question to answer is whether there are existing mechanisms, e.g. in programming languages which can be integrated and advanced on a model level. Beyond that other aspects could be considered such like model transformations. The goal is to establish an easy reuse of meta models, or parts of them, to enable time-efficient and error-minimizing design of meta model families.

In the following Section 2 some related work is presented which served as inspiration to this research and ideas for possible solutions. Section 3 gives a brief introduction to the CINCO Meta Tooling Suite and how model reuse is implemented today. This leads to reoccurring problems which are described in Section 4. In Section 5 we introduce a more potential solution for model reuse in CINCO to overcome these problems. Finally Section 6 concludes presented results and gives some outlook to our future work.

2 Related Work

A different but very related problem to reuse implementation effort is the famous expression problem [Wad98], which addresses the question on "how to extend a program?". Usually an efficient solution is requested and efficiency is understood in terms of source code complexity.

The problem can be extended by limitation of recompiling or redeployment of old program parts. Although the problem came up at the end of the last century, research for solutions never stopped [Tor04, WO16, LH06].

Furthermore, pure::variants[1] has to be mentioned as a software, which already provides development of product lines in an integrated environment but only on a source code level.

Leaving the technical level of source code, some published results can be found in the sector of MDD. The concept of archimedian points [SN16] addresses the aspect of transforming a new variant of a meta model to an older version to enable reuse of meta model languages and corresponding code generators. The idea of splitting the meta model up into fragments and reusing them as extensions in other ones is proposed in [RVMS12].

Besides these general design concepts, there are at least two more practical approaches which define and implement SPLs in MDD. [DSW14] covers a more declarative way to distinct variable features by only describing their differences. In [GV09] the paradigm of aspect oriented programming is applied on the model-level to propose not only a declaration, but also a model-to-code transformation to implement software variants.

Development of product lines not only results in the question of extensibility as part of variant management. The paradigm of higher-order process modeling in the context of MDD, which is researched in [NSM13], can also provide mechanisms to improve the flexibility of meta model declaration.

3 Meta Tooling with CINCO

The CINCO Meta Tooling Suite [NLKS17] offers a development environment for graphical modeling tools. Based on textual specifications a complete Eclipse-RCP-based Integrated Modeling Environment (IME) is created by full code generation. These textual specifications consist of the Meta-Graph-Language (MGL) to define the meta model and a description of appearances in the graphical editor by the Meta-Style-Language (MSL).

In addition, the IME can be enriched with model validation, which is either generated from the meta model or manually implemented. To support development on tool level CINCO offers integration of code generators for specified meta models. This allows to create domain-specific graphical modeling environments only by a few lines of specification.

Listing 1 gives an example specification of a simple meta model for control flow. A model has a name and can contain nodes of types `Start`, `Activity` and `End` of which `Start` nodes are limited to exactly 1 and `End` nodes to at least one. All of these are connected by edges of type `Transition`. The model elements `Activity`, `End` and `Transition` are labeled. A sample model is shown in Figure 1.

A proper addition to the meta model would be a possibility for model hierarchy. This enables a first level of model reuse similar to procedural programming on code level. To reuse a model a "procedure call" has to be defined and used. Because models in general do not always represent any sort of control flow these "calls" are described by representational model elements.

From a graphical modeling perspective, the creation of a reference is initiated by dragging a specific model or any of its inherent elements and dropping it on the active workbench. In

---

[1] www.pure-systems.de

```
1   graphmodel ControlFlow (
2     attr EString as name
3     containableElements(
4       Start(1,1),Activity(0,*),End(1,*)
5     )
6     node Activity (
7       attr EString as label
8       incoming(Transition(1,*))
9       outgoing(Transition(1,*))
10    )
11    node Start (
12      outgoing(Transition(1,1))
13    )
14    node End (
15      attr EString as label
16      incoming(Transition(1,*))
17    )
18    edge Transition (
19      attr EString as label
20    )
21  )
```
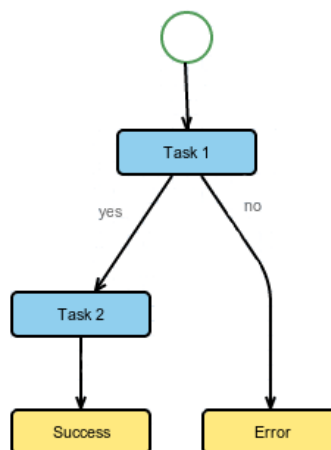
Listing 1: ControlFlow.mgl

Figure 1: A sample ControlFlow

general, the reference to another model needs to be represented in the current model in some way. The CINCO framework supports the automatic registration of a drop handler to be called whenever such a drag-and-drop action is performed.

This drop handler represents the junction point between model references and defined semantics as it typically triggers code for creation of the model elements that represent specific artifacts. These may span the model itself, any of its inherent elements as well as any other artifact that is somehow associated with any of these.

To model a switch into another model and handle possible outcomes the meta model is extended with an additional node and edge type. Listing 2 shows necessary additions on the meta level. A new `SubModel` node is introduced which represents the switch of control to the start node of another model. First, it inherits from `Activity` and therefore has a label and incoming `Transitions`. Second, it overrides outgoing edges to type `Branch`. A new edge type is introduced to represent possible outcomes of the underlying control flow model, which reference to according `End` nodes in the submodel symbolically.

Third, a `prime` attribute is defined of which a model element can only have one. This is the actual reference to another model element and establishes a link between these. In this case the attribute `submodel` references the other `ControlFlow` model. Other specifications are also imaginable: The `prime` reference could point directly to the `Start` node and additionally `Branch` edges could have been referenced by a `prime` attribute to an `End` node.

In Figure 2 the control flow of Figure 1 is reused as follows: The `SubModel` node references the reused `ControlFlow` model in the `prime` attribute. The possible multiple outcomes in form of `End` nodes are represented by outgoing edges of type `Branch` and labeling them accordingly.

The simple example shown in Listing 2 and Figure 2 shows how references between model

```
1   ...
2   node SubModel extends Activity (
3     prime ControlFlow as submodel
4     outgoing(Branch(1,*))
5   )
6
7   edge Branch extends Transition (
8   )
9   ...
```

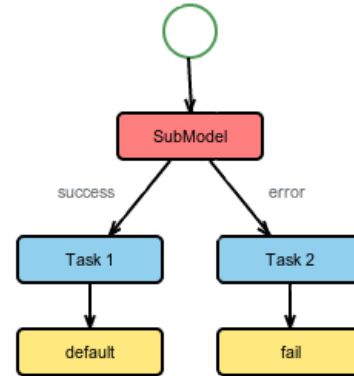Listing 2: Extending ControlFlow.mgl

Figure 2: Sample of model hierarchy

elements can be defined in CINCO meta model specifications. Note that it is possible to even define references across instances of different meta models. Potential problems as well as possible solutions in this context are presented in the following sections.

4 Solution Approaches and Recurring Problems

References to some artifact, e.g. to another model, need to be somehow represented in the current model. This usually means presence of a set of specific model elements, each of which representing the referenced artifact itself or another artifact that is somehow associated with the referenced one. However, as these representational model elements share a common relation to the referenced artifact they form associated components, as a whole referred to as a *compound* throughout the following discussion. Handling these compounds on the model level comprises the following aspects:

- **Initialization**: As soon as the respective reference has been created the compound needs to be initialized by means of creating all associated model elements and setting up their attributes.

- **Updating**: The compound is to be updated if the referenced artifact changes, i.e. refactoring to rebuild structural integrity - be it automatically or in a manual manner.

- **Validation**: The structural integrity of the compound should be checkable to generate a warning or error if incorrect.

- **Deletion**: The compound should be deletable by means of deleting all associated model elements at once.

Before a structured approach to the realization of these aspects is described, it should be mentioned that in practice we found that the realization of these aspects often is not pursued in a uniformly structured manner. One reason for this are developers that follow naive approaches

without a view of the whole. Another reason is intentionally independent development effort due to separation of concerns. As an example, the validation of models might be developed independently from the structural logic or graphical representation of the model elements. However, along with a potential lack of consultation between the developers in the worst case this might lead to multiple separate implementations for each of these aspects with scattered semantics due to distributed logic as well as a substantial amount of code duplication.

A structured approach to the realization of the compound-related aspects can be achieved with written code that relies on a suitable data structure. The latter should reflect both the inherent structure of a specific compound as well as the underlying reference to the respective artifact. The logic that covers each of the compound-related aspects (initialization, updating, validation and deletion) can build upon this data structure. This logic only slightly differs depending on each respective aspect:

- **Initialization** means creating all necessary model elements associated to a compound according to the current state of the referenced artifact.

- **Updating** first of all requires the identification of already existing compound-related model elements. These need to be synchronized with the model elements that should exist according to the current state of the referenced artifact. Obsolete elements need to be deleted while missing elements need to be created. Note that it is not an efficient solution to completely remove and re-insert the model elements associated with a compound because existing ones might have been customized by the user and deletion as well as creation might trigger additional model-transforming routines that probably should not (again) be triggered.

- **Validation** - like updating - requires the identification of already existing compound-related model elements. But instead of restoring structural integrity, it is only checked by means of comparing the model elements that exist to those that really should exist according to the current state of the referenced artifact.

- **Deletion** also rests upon the identification of existing compound-related model elements.

Altogether, the realization of each of these compound-related aspects in total makes up the synchronization logic. It can be defined in a generic manner as it is independent of the type of elements that actually are handled. In particular, it can be separated from the structural information about a specific compound as it all comes down to identifying and comparing model elements related to a compound. However, the implementation requires the existence of routines to compare, add and delete model elements as typically provided by the modeling framework, e.g. by Cinco .

The focus lies on reusability of compounds, and as already mentioned above a suitable data structure reflects the structural information about a compound, i.e. it defines the referenced artifacts for each compound-related element and provides a matching routine to determine whether an existing model element represents a specific artifact. Applied to the example shown in Listing 2 and Figure 2, the data structure holds the information that the compound consists of a `SubModel` node and various branches, i.e. edges of type `Branch`. Additionally, it holds the information that each branch represents an `End` node of the referenced model whereas the

`SubModel` node somehow represents the referenced model itself. A corresponding matching routine would check whether a suitable `Branch` edge exists for each represented `End` node, i.e. the edge exists and its label equals that of the respective `End` node.

In this example, the compound consists of a `SubModel` node and `Branch` edges. However, the focus lies on reusability of compounds and the actual type of represented artifacts may depend on the concrete reuse scenario. Hence, the fact that the artifacts represented by `Branch` edges are the `End` nodes of a referenced model is characteristic of the implementation in terms of the specific use case of this example. This circumstance can be reflected in code. Consider an abstract data type `SubModelCompound` to hold abstract methods `getSubModel` that retrieves the referenced model as well as a method `getBranchReferences` that retrieves the artifacts to be represented by `Branch` edges. The reuse of the compound would be achieved via multiple concrete implementations of the abstract data type, each of which pointing towards a specific type of model as well as specific type of branch references. This requirement can be tackled by means of type parameters in the context of generic programming.

The depicted approach based on written code is scalable to an arbitrary number of model elements that make up the actual compound. From an abstract point of view, the depicted solution - though it is implemented in source code - already has a declarative tendency, because in essence the concrete parts of an implementation link representational elements to those that are actually represented. The logic regarding initialization, updating, validation and deletion can be defined on abstract types and is independent of the concrete reuse scenario. This observation leads to ideas for a realization on a higher level of abstraction to be described in the next section.

5 Language Improvements via Linking Model

In the previous section reoccurring problems were presented in managing model elements and their representations and also a solution to this on source code level was introduced which made use of a general description of the relation between original and representational elements. Although this works fine for the moment this idea still has some essential drawbacks on a practical level.

One is that the description of how something is represented with model elements is hidden on source code level from the domain expert. Without programming knowledge, it is not possible for him to change the behavior of the modeling tool when creating such representational elements. Instead, this could also be understood as a model transformation from original elements to some representation, which leads to the various different ways to describe such one and some of them are likely known by domain experts. This gives them the possibility to specify application semantics with more detail by themselves.

Another problem relates to the strict linking between origin and representation by the existent `prime` attribute. The solution allows to reuse models or model elements, but some kind of reuse for the representation is denied. The model transformation implemented on source code level does only handle structural information. But because it creates a graphical representation there are more aspects that need to be considered, like creating a desired layout for these model elements.

An example to a more complex transformation in the context of the DyWA Integrated Model-

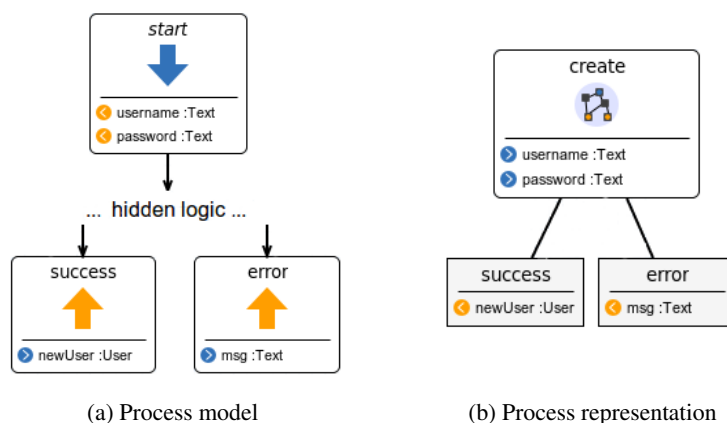(a) Process model · (b) Process representation

Figure 3: Reuse of process models in DIME

ing Environment (DIME) [BFK⁺16] is given in Figure 3. DIME is a CINCO product for specification and generation of web applications. On the left side a shortened Process model of DIME [BFK⁺16] is shown. A Process in DIME is much like the control flow in Figure 1 but with added support for data flow aspects. As only the start and ends of such model are relevant for the discussion, the actual process logic in between is not show in the figure. It translates to its representation shown on the right side. A Process not only has a start and multiple ends, but these also have so called ports which represent input parameters and return values.

Besides application behavior the user interface has also to be modeled. Therefore, it offers the meta model GUI which has to be integrated in process models to enable manipulation by program control flow. By design, to have a somewhat consistent look of process models, the representation of GUI models in a process is very similar to the one shown in Figure 3. Although the transformation of a GUI model in a process on a structural level is very different, the graphical layout for the representation should be reused both for Process and GUI models. Therefore, it is necessary to decouple the strict linking of representational model elements by the `prime` attribute in a MGL to their origins.

The solution proposed in this paper centers around expanding the CINCO language family by a new meta model specialized on loose linking of model elements to replace the existing `prime` attribute. To satisfy the needs the new meta model has to offer a minimum set of properties to describe a reference:

Source It must be possible to define to which model or model element something is referenced to. On tool level this describes which object can be dragged on the modeling canvas.

Container Possible areas need to be specified, where dragged objects can be dropped. In the context of CINCO this means a set of container model elements.

Target The *compound* model element links to the source and can be reused in multiple scenarios. The actual definition of such *compound* will still be done in the MGL model.

Handler A description of how the source is translated to the target is needed. This basically replaces the source code solution of Section 4 and extends to the possibility to use any transformation language. The linking between model elements is decoded in the transformation and can be dynamically created on runtime.

Applied to the example of Listing 2 the `prime` attribute is removed and instead one description tuple of loose linking is created. Based upon this information the familiar tool behavior can be maintained. CINCO uses the source and container information to call the appropriate handler in the form of a transformation when a control flow is dropped on the canvas. The transformation outputs a structure of a `SubModel` node with connected `Branch` edges. Additional functionality like layout-creation is separated from the transformation and could therefore be reused whenever elements of the target *compound* are created.

With this model the translation semantic of model elements to their representation could possibly be done by a domain expert. If then another meta model is integrated e.g. in this control flow model, a second tuple of loose linking has to be created, which includes writing a specialized transformation for that case. But this could purely be done again by the domain expert as it is not hidden in the source code. Also representational element definitions and related layouts are reused automatically and because the creation of representations and keeping them synchronized with their origin is generalized the benefits of the solution discussed in Section 4 still apply.

6 Conclusion

In this article we proposed a solution for specification duplication in the context of inter-model references. Exemplary shown for the CINCO Meta Tooling Suite the idea of a loose linking model has been introduced, which supports domain experts in their specification of modeling tool behavior and meta model description. The new model separates the concerns of model definition and tool behavior by means of shifting the specification of references and corresponding representations from the source code to a declarative language, hence making them accessible for domain experts without programming knowledge. Additionally, it enables the reuse of representational *compound* elements for different purposes. It facilitates syntactical correctness in terms of valid references and thus reduces implementation effort. However, basic tool features like layout-creation for now remain on code-level. Applying the same level of abstraction on these aspects might be a topic of future work.

The depicted idea to enhance the CINCO Meta Tooling Suite towards optimizing development time of software product lines is only one out of many. The next steps will address the implementation of an actual linking model as well as the evaluation of the benefits on a more sophisticated product like DIME. Beyond that, more concepts of modern programming languages like polymorphic dispatching could be pushed up to the modeling level, especially in terms of connecting and reusing models. The goal is to maximize the flexibility on all levels of modeling, from the domain expert as a tool developer to the actual modeler, to enable the specification of products and product variants without the need for a programming expert to be present.

Bibliography

[BFK⁺16]   S. Boßelmann, M. Frohme, D. Kopetzki, M. Lybecait, S. Naujokat, J. Neubauer, D. Wirkner, P. Zweihoff, B. Steffen. *DIME: A Programming-Less Modeling Environment for Web Applications*. Pp. 809–832. Springer International Publishing, Cham, 2016.
doi:10.1007/978-3-319-47169-3_60

[BNNS16]   S. Boßelmann, J. Neubauer, S. Naujokat, B. Steffen. Model-Driven Design of Secure High Assurance Systems: An Introduction to the Open Platform from the User Perspective. In T.Margaria and M.G.Solo (eds.), *The 2016 International Conference on Security and Management (SAM 2016). Special Track "End-to-end Security and Cybersecurity: from the Hardware to Application"*. Pp. 145–151. CREA Press, 2016.

[BPS04]   D. Beuche, H. Papajewski, W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming* 53(3):333–352, 2004.
doi:10.1016/j.scico.2003.04.005

[DSW14]   F. Damiani, I. Schaefer, T. Winkelmann. Delta-oriented multi software product lines. *Proceedings of the 18th International Software Product Line Conference on - SPLC '14*, 2014.
doi:10.1145/2648511.2648536

[GV09]   I. Groher, M. Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. *Lecture Notes in Computer Science Transactions on Aspect-Oriented Software Development VI*, p. 111–152, 2009.
doi:10.1007/978-3-642-03764-1_4

[KT08]   S. Kelly, J.-P. Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[LH06]   A. Löh, R. Hinze. Open data types and open functions. *Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '06*, 2006.
doi:10.1145/1140335.1140352

[LNS13]   A.-L. Lamprecht, S. Naujokat, I. Schaefer. Variability Management beyond Feature Models. *Computer* 46(11):48–54, 2013.
doi:10.1109/mc.2013.299

[MS09]   T. Margaria, B. Steffen. Business process modelling in the jABC: the one-thing-approach. *Handbook of research on business process modeling*, pp. 1–26, 2009.

[NLKS17]   S. Naujokat, M. Lybecait, D. Kopetzki, B. Steffen. CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools. *STTT*, 2017. to appear.

[NNL+13] S. Naujokat, J. Neubauer, A.-L. Lamprecht, B. Steffen, S. Jörges, T. Margaria. Simplicity-first model-based plug-in development. *Softw. Pract. Exper. Software: Practice and Experience* 44(3):277–297, 2013.
doi:10.1002/spe.2243

[NNMS16] S. Naujokat, J. Neubauer, T. Margaria, B. Steffen. *Meta-Level Reuse for Mastering Domain Specialization*. Pp. 218–237. Springer International Publishing, Cham, 2016.
doi:10.1007/978-3-319-47169-3_16

[NSM13] J. Neubauer, B. Steffen, T. Margaria. Higher-Order Process Modeling: Product-Lining, Variability Modeling and Beyond. *Electron. Proc. Theor. Comput. Sci. Electronic Proceedings in Theoretical Computer Science EPTCS* 129:259–283, 2013.
doi:10.4204/eptcs.129.16

[RVMS12] D. Ratiu, M. Voelter, Z. Molotnikov, B. Schaetz. Implementing modular domain specific languages and analyses. *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation - MoDeVVa '12*, 2012.
doi:10.1145/2427376.2427383

[SM09] B. Steffen, T. Margaria. Continuous Model Driven Engineering. *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, 2009.
doi:10.1109/iceccs.2009.58

[SN16] B. Steffen, S. Naujokat. *Archimedean Points: The Essence for Mastering Change*. Pp. 22–46. Springer International Publishing, Cham, 2016.
doi:10.1007/978-3-319-46508-1_3

[SSA14] C. Seidl, I. Schaefer, U. Aßmann. Integrated management of variability in space and time in software families. *Proceedings of the 18th International Software Product Line Conference on - SPLC '14*, 2014.
doi:10.1145/2648511.2648514

[Tor04] M. Torgersen. The Expression Problem Revisited. *ECOOP 2004 – Object-Oriented Programming Lecture Notes in Computer Science*, p. 123–146, 2004.
doi:10.1007/978-3-540-24851-4_6

[VV11] M. Voelter, E. Visser. Product Line Engineering Using Domain-Specific Languages. *2011 15th International Software Product Line Conference*, 2011.
doi:10.1109/splc.2011.25

[Wad98] P. Wadler. The Expression Problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt/, Nov 1998. [Online; accessed 13-July-2017].

[WO16] Y. Wang, B. C. D. S. Oliveira. The expression problem, trivially! *Proceedings of the 15th International Conference on Modularity - MODULARITY 2016*, 2016.
doi:10.1145/2889443.2889448