Position Paper: Towards a Minimalistic Modeling Paradigm

Gábor Kövesdán, Márk Asztalos, and László Lengyel

***Abstract*—Model-Driven Software Engineering (MDSE) has put a great emphasis on modeling to deal with the rapidly growing complexity of software systems. This leads to that models become more complex and detailed as they evolve. In contrast to the vision that such models are able to deal with this complexity, there are several cases in which simple models better support the requirements. In this paper, three technology trends are presented. The success of these trends lays in using lightweight models. Of course, inherently complex systems tend to use complex models but the main goal is to focus modeling on supporting requirements and avoid adding unnecessary details. After the explanation of the technology trends, their key success factors are identified. By these success factors, we outline some considerations that facilitate maintaining the complexity of models low.**

***Index Terms*—Modeling, model-driven software engineering, agility.**

I. Introduction

Model-Driven Software Engineering (MDSE) [1], [2] is a widely applied method in software engineering, which uses the model as a first-class artifact in the engineering process. The model is leveraged in software development instead of focusing on algorithmic concepts. The *Model-Driven Architecture (MDA)* methodology, standardized by the *Object Management Group (OMG)* even tries to avoid programming in the conventional sense and use modeling instead as a primary means of software development. The expansion of these technologies has emphasized the importance of the model. Nowadays, there is a big effort in creating more accurate and complete models. Often we ask ourselves, how we efficiently model problem domain *D* to capture all of the significant details.

At the same time, complexity of software systems grows at a rapid pace. The result of this is that models become always bigger and more complex. Their purpose is to facilitate software development but big and complex models are very rigid and make future changes difficult. They also have a negative impact on scalability [3]. There is ongoing research on how to deal with distributed models and parallelized model transformations [4] since dealing with such a high complexity is otherwise not possible. It is evident that complex systems also require complex models. However, these problems could be mitigated at their roots by limiting models to the minimum extent that is needed for the requirements.

This paper proposes reconsidering our approach towards modeling and simplifying models when it is possible. We believe that the essential purpose of models is to support the requirements. Thus the right question is not how to model problem domain *D* but how to support feature *F*? Taking this into account, the MDSE term does not grasp the main concern in modeling since it only suggests that the software engineering process is driven by models but it ignores the importance of requirements. In fact, modeling should be driven by the requirements and later engineering phases by both requirements and models. This is similar to Agile software development [5] combined with modeling. If models are created without a special focus on requirements, it is harder to maintain them simple, avoiding details that are not necessary for the requirements. More specific suggestions on this will be provided at the end of the paper.

This paper presents three technology trends of a general nature where simpler models have proven to be more efficient in practice. These trends are well-known and have created numerous technology debates. Nevertheless, extracting the conclusion of how to keep models simple is our own contribution. Despite that the description of technology trends is not so detailed as concrete case studies, they suggest some lessons to learn. New technologies would not become widespread if they were not any better than older ones. By observing technology trends we can find valuable conclusions that are worth paying attention to.

The rest of this paper is organized as follows. Section II describes how the inherently simpler and more limited *domain-specific languages (DSLs)* [6]-[8] can achieve a higher increase in productivity than detailed and complete models. Section III compares the relational data model to Neo4j's [9] graph data model. The first one does not efficiently support connections since it totally lacks the notion of relationships. Besides, it uses strict schemas to enforce the structure of persisted data, which makes later changes difficult. In contrast, the latter supports relationships and their efficient traversal and by having a schema-free nature, it facilitates easy refactoring of the model. Section IV describes SOAP and RESTful Web Services and why the simpler data model of the second one has become more successful. Section V summarizes the lessons to learn from these trends. Section VI concludes the paper.

Manuscript received September 30, 2013; revised November 23, 2013. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred. This work was partially supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR_12-1-2012-0441).

The authors are with the Budapest University of Technology and Economics, Department of Automation and Applied Informatics, Budapest, Hungary (e-mail: gabor.kovesdan@aut.bme.hu, laszlo.lengyel@aut.bme.hu, mark.asztalos@aut.bme.hu).

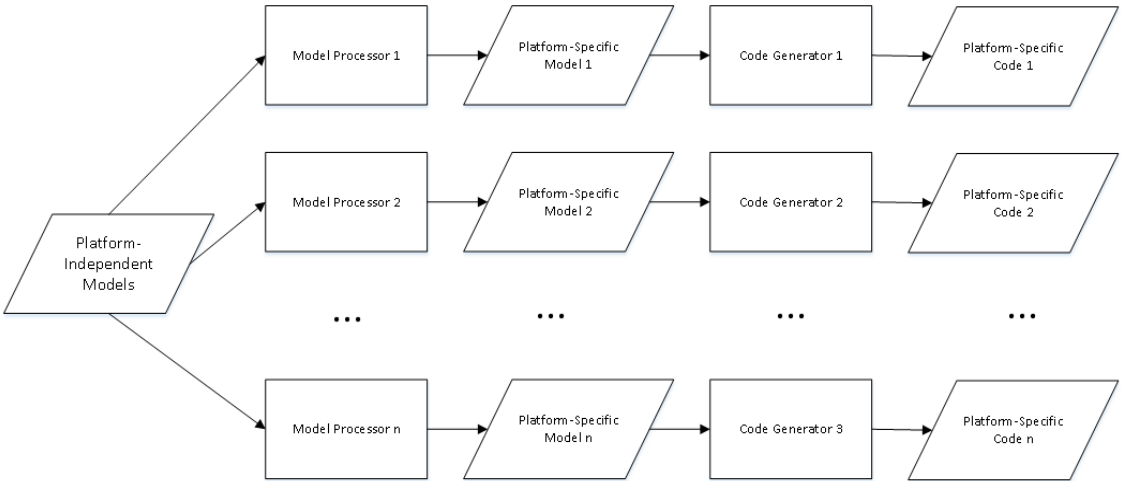

Fig. 1. The MDA development workflow.

II. UML, MDA and Domain-Specific Languages

The Object Management Group (OMG) provides a full-fledged general metamodeling standard with *Meta Object Facility (MOF)* [10], which also became an international standard. It provides a four-layer metamodeling architecture and falls into the category of strict metamodeling architectures, that is, elements of a specific layer have strict correspondence with a model element of the previous layer. MOF defines the topmost metamodeling layer, which conforms to itself. This layer can be used to describe modeling languages. *Unified Modeling Language (UML)* [11]-[13] is an example of such a language. UML is a general-purpose modeling language that can be used to model any type of problems from a software developer's point of view. It has several diagram types to describe both structure and behavior.

Based on the above two standards, OMG created a model-based development architecture, called *Model-Driven Architecture (MDA)* [14]. MDA uses the UML standard for modeling software systems. Its main concept is that a *Platform Independent Model (PIM)* is created to model those aspects of the application that do not depend on any particular technology or target platform. The PIM is later translated to the *Platform Specific Model (PSM)* that, in turn, models a system targeted to a specific runtime environment. The translation to PSM is a model transformation that accomplishes the required mappings to constructs in the target technology. This transformation must be modeled as well. The PSM can be considered an executable artifact since it models the final system on the top of a concrete technology stack. Figure 1 shows the development workflow of MDA. The MDA vision is that a suitable tooling can properly accomplish this translation and the whole development process of the system is done by modeling, without having to write program code. In other words, MDA uses UML as a *general-purpose programming language* (GPL) [11]. Another effort to use UML as a GPL is *Executable UML* [15].

Although there are case studies that show how MDA works in practice to increase the quality and the productivity [16], according to Fowler's vision [11], using UML as a programming language is not a realistic solution. First, the success depends on the required complex tooling. Secondly, even if using UML as a programming language is possible, it is not proven to be significantly more productive than another programming language. Kelly and Tolvanen also report that using UML do not significantly increase productivity when compared to domain-specific solutions [6].

According to Fowler, the most efficient use of UML is using it for sketches. UML is very practical to document design decisions and facilitate communication between developers. These sketches only cover a highlighted part of the system that is being documented and omit finer-grained details that are not relevant for the purpose of the sketch. Summarizing the uses of UML modeling, we can conclude that the comprehensive and complete approach do not usually increase productivity in a significant manner, whereas a minimalistic approach still offers the major advantages of modeling without introducing any hassles into the engineering process.

Domain-Specific Modeling Languages (DSLs) represent another direction in modeling, very different from UML. DSLs have a wide range of uses [8] but they inherently have a limited scope. Their common characteristics are that they have a limited expressive potential with which they can only express problems of a well-defined problem domain. This means that DSLs only model what is really necessary for supporting requirements. They do not aspire to be complete at all. Sometimes, a DSL modeling environment is composed of several DSLs, using each of them for modeling different aspects of the system. Several use cases have shown [6] that unlike UML and general-purpose modeling, DSLs significantly increase productivity and software quality. DSLs are thus another example of how the simplification of the model facilitates software engineering. The multi-mobile platform developed at the Department of Automation and Applied Informatics of the Budapest University of Technology and Economics[1], is also a good example of how a DSL can be used for increasing productivity. The platform uses a DSL to model smartphone applications and generate source code for several mobile platforms. One of its most important features is that it does not try to model everything, just the commonalities of the different mobile platforms. This means that writing source code is still necessary, the platform does not try to avoid it. On one hand, the business logic can be usually expressed more easily with ad-hoc source code

[1] Publication of the results is in progress.

rather than with modeling. On the other hand, providing a complete and rigorous modeling environment would not be flexible because of the diversity of different mobile platforms. Modeling such a system as a mobile application would be much more difficult with UML or any general-purpose modeling language.

III. The NOSQL Movement

Until recently, database solutions have used almost exclusively the *relational data model.* In the relational data model, data is decomposed into tables. Each table encapsulates related properties, which are represented as table columns. Each column has a well-defined type. Occurrences of the property set must conform to these types and are represented in different rows, also called records. This means that the relational data model defines a strict schema for the data to be stored.

To review how the relational data model can be used to model real-life problems, we will use the entity-relationship (ER) model [17] as a reference. The ER model is very close to human thinking. This makes it useful to use it as a reference data model to examine how well other data models can express domain notions.

Domain entities of ER models are usually mapped to tables. Connections do not fit very well into the relational model since they lack the notion of relationships. A relationship can only be represented by a reference to a unique identifier of the connecting record. In one-to-one relationships, a column of any or both ends can hold the reference to the identifier of the connected record. Many-to-one relationships can still easily be mapped by having the reference in records of the many side. Nevertheless, many-to-many relationships require the introduction of an additional connection table that lists the identifier pairs of connected records.

Apart from this mapping, relational database schemas are usually normalized to avoid redundancy and its unwanted effects. The normalization process creates more tables by decomposing sets of columns into new tables. These phenomena create a gap between the relational and the ER data models. This also means that the relational model does not match the human way of thinking about the problem domain. The strict schema and the divergence of the intuitive ER data model make the relational model rigid and resistent to changes.

An even more serious problem with the relational data model is how connections are handled. If such data are queried that are decomposed among several tables, tables must be joined in order to obtain the result. The joint operation of the relational model first constructs the Cartesian-product of table rows of the joined tables and then only maintains the resulting rows where corresponding records have been combined. This filtering is done by checking the identifiers and corresponding references. If the tables that are being joined have m and n rows respectively, the Cartesian-product will have $m \times n$ rows. So the join operation has a high inherent complexity, which may sum up when performing several joins in a single query. The number of computation steps is exponential in at least the minimum row count of the joined tabled. This means that the data model does not support efficient queries and data manipulation on big amount of data. Relational databases usually leverage some optimizations through indexing and caching, which will mitigate the performance hit of join operations but the cost is inherently high in the general case, so joins will necessarily affect performance in large systems.

These disadvantages urged new data models to emerge. Collectively, these are called NoSQL data models. All of them try to address these weaknesses. In this paper, we present the Neo4j [18] graph database as an example. There are some variations in the data model that graph databases use. Neo4j uses the most widely applied variant, the *property graph data model.* It deals with nodes and directed edges, allowing properties on both. In Neo4j databases, connections are first-class artifacts thus the data model does not have to diverge from the entity-relationship model.

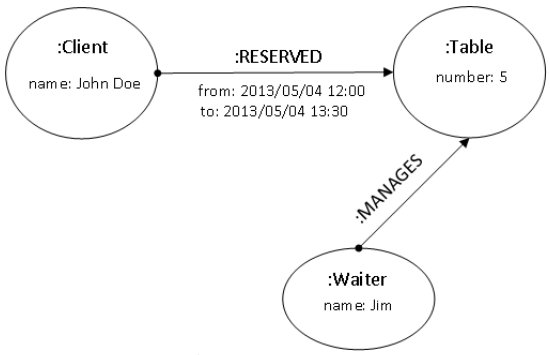

Fig. 2. Model of table reservations in a restaurant.

Fig. 2 shows a model of table reservations in a restaurant. Clients are connected to reserved tables and the connection holds the date of reservation. Waiters are connected to the tables they manage. We can also observe the properties of the entities.

Neo4j databases are totally schema-free. Node and connection types are supported with the notion of labels but they do not mean strict typing since they do not constrain the properties and edges the node may have. In practice, this is not a problem since applications written in object-oriented (OO) languages usually map nodes and connections to objects and object references. If the mapping is considered to be a validated model transformation, the typing of OO classes guarantees that no meaningless information is stored in the database. In practice, these mappings are not validated but a well-tested and mature mapping framework enforces type safety. If extraneous properties or connections are still added in some unforeseen way, the mapping can ignore them. Mapping of Neo4j databases to Java objects is implemented by Spring Data Neo4j [19]. So the mapping solves type safety and allows for using a schema-free, lightweight data model. This data model provides for much higher flexibility than the relational data model, since relationships are first class artifacts and do not require join operations or referencing properties. This means that relationships can easily be added in a later development phase. The schema-free nature also allows adding new properties to individual nodes. Existing applications can silently ignore these extraneous nodes until they are prepared to use them. This allows for incremental software development.

Unlike relational databases, the Neo4j graph database supports efficient queries on connected data. A query consists of the following three steps:

1) Finding potential start nodes that satisfy the specified

conditions.
2) Pattern matching in the graph, using the start nodes as reference points.
3) Returning the result, based on matching subgraphs.

Finding the start nodes is equivalent to finding rows in a relational database table. Both data models support indexing that allows for fast retrieval of nodes or rows based on the indexed properties. Similarly, the third step is also equivalent in the relational data model. The result retrieved in the second step is either presented as is or an additional filter or aggregation is applied.

The main difference is in the second step. As seen before, following a connection in the relational data model means as many computational steps as the product of row counts of joined tables. In Neo4j's property graph data model, a connection is a simple reference and following it means constant computational steps. So traversing a subgraph has linear complexity in the number of traversed nodes.

In summary, the Neo4j graph database supports better extensibility and more efficient queries [20], [21] of connected data. Both are consequences of its lightweight, schema-free data model. The first is a direct consequence and the second is a result of the connection support and the lack of scattering the information as done in relational databases during normalization. Relational databases make a great effort on supporting distributed computing [22] so that big database systems can scale to large amount of data. Clustering support is also indispensable in graph databases but they may require less servers in total since they are inherently more efficient.

From the above reasons, it can clearly be seen that graph databases have some significant advantages when compared to relational databases. These advantages all come from the simpler data model so this case is also a premonitory to consider minimalistic models.

IV. SOAP Versus Restful Web Services

Simple Object Access Protocol (SOAP) [23] is a remote messaging framework standardized by the *World Wide Web Consortium (W3C)* that can be used to exchange structured information in computer networks. *Simple Object Access Protocol (SOAP)* messages use an *eXtensible Markup Language (XML)* vocabulary to address messaging endpoints and execute operations. The enclosed messages are also described with XML, whose schema is defined by the endpoints. SOAP endpoints use the *Web Service Description Language (WSDL)* [24] to specify their interfaces. The message schema of parameters and return values are also part of the interface. This interface description may be obtained from a Universal Description, Discovery and Integration (UDDI) repository or from a vendor-specific source. The above outlined architecture of XML Web Services is demonstrated on Fig. 3.

By using XML syntax, SOAP provides for a platform- and vendor-independent solution to achieve integration with remote services. SOAP supports representation of complex data structures, routing, synchronous and asynchronous messaging and conversational state. However, these features have a high cost. The complexity of SOAP and the related WS-* standards make implementations difficult. Besides, parts of the specification are unclear, which caused interoperability problems despite that the main objective of the Web Services standard family was technology-independent interoperability. This urged another recommendation, called *Web Services Interoperability (WS-I),* which defines interoperability criteria. Another problem with SOAP Web Services is that because of conversational state, SOAP does not scale well.

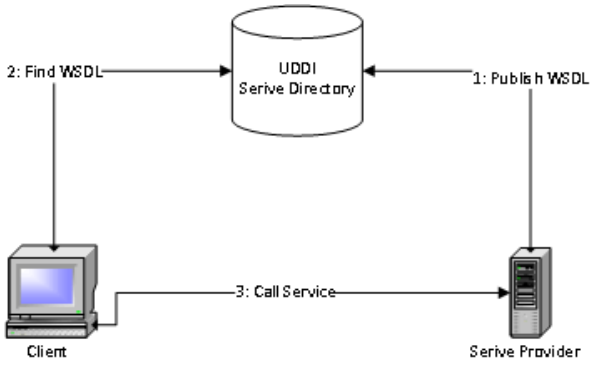

Fig. 3. The architecture of XML web service.

The *Representational State Transfer (REST)* [25] architectural style has become a new trend for avoiding the complexity of SOAP Web Services and to remedy some of their disadvantages. REST itself is not a Web Services standard, just an architectural style but Web Services can be implemented in a REST style. Representational state transfer means that resources are not accessed directly but are queried and manipulated via their representations. The REST architectural style is not bound to any specific protocol but is usually implemented with the *Hypertext Transfer Protocol (HTTP),* which identifies resources with URIs and provides HTTP methods to perform operations on them. HTTP is thus a suitable protocol to implement Web Services in a REST manner, or how it is usually called among developers, in a Restful way [2]. REST operations are not arbitrary, they must fit into the verb-noun abstraction, for example, read resource *A*, update resource *B*, and so on. These are mapped to the corresponding HTTP requests, like *GET* and *PUT*. This abstraction is very suitable for supporting Create-Read-Update-Delete (CRUD) operations. In REST, there is no communication state, just like HTTP is inherently stateless as well. Thus resource state representations are always included in requests. These representations are application-specific and a single application may support several different formats. Usual representation formats are XML and *JavaScript Object Notation (JSON),* being the second one more common. JSON is schema-free and RESTful Web Services do not usually have a strict specification either. There are initiatives [26] to create interface descriptions, like WSDL descriptions of SOAP Web Services but they are not commonplace.

HTTP and JSON together make up a suitable technology stack for creating simple, flexible and scalable remote interfaces. RESTful Web Services definitely lack several features that are present in SOAP, for example, stateful messaging, complex message exchange patterns, and partial encryption of messages and so on. Besides, REST is more appropriate for finer-grained operations and complex course-grained methods do not fit well into the abstraction. Nevertheless, RESTful Web Services are gaining in popularity at the expense of SOAP Web Services. Companies

that provide their services in both styles report that RESTful Web Services are preferred by their clients [27]. REST is a good example of that a simpler model can be more efficient than a more detailed one.

V. Designing Minimalistic Models

We believe that the outlined concepts and the explained technology trends serve as a good basis for reconsidering our approach towards modeling. By studying these trends we can learn the following points:

1) *Concentrate on the requirements:* The modeling approach that we use in a particular software system should be determined mostly by the requirements rather than by general considerations. For example, if the software uses connected data, it would probably benefit from using a lightweight graph database. Despite that graph databases are usually schema-free and thus, as opposed to relational databases, do not enforce consistency, they support efficient traversal of connected data. In turn, relational databases do not handle well connections but are powerful with aggregated data.
2) *Plan what models will be used for:* Models can be used in several different manners. Sometimes it is easy to model a problem and generate code than developing the system in a purely programmatic way. In other cases the model serves as a contract, which is later used for validation. A different scenario will leverage models to facilitate communication with domain experts. It is important to plan what the model will be used for since doing so may allow for the omission of unnecessary abstractions that are not required for the use cases. A part of this question is how models are constructed. If it is possible to reason about the synthesis of models, the actual metamodel do not have to enforce all of the validity constraints.
3) *Consider partial modeling:* Like other methods, modeling is not a silver bullet. It is completely acceptable to use modeling for some parts of a software system but omit modeling for other parts, where it does not help. This is also what the above mentioned multi-mobile platform achieves. Only the commonalities are modeled that are similar in all mobile applications but business logic is developed with ad-hoc programming since it is too general to leverage modeling. Besides, lightweight models are easier to enrich later than simplifying complex models. The *Multi-Paradigm Modeling (MPM)* approach also shares these ideas [28].
4) *Invest in customized solutions:* Although standardization in modeling brings several advantages to the software product like reuse, interoperability, improved learning curve, a customized solution can increase more the productivity than a Swiss army knife-like solution. Domain-specific modeling allows for working with the notions and at the abstraction level of the problem domain, as opposed to general-purpose modeling, which requires the developer to establish a mapping between notions of the domain and modeling elements. Current modeling tools enable developers to easily create domain-specific model editors and run transformations, like code synthesis on them.
5) *Leverage software layers:* Most software systems are layered. This allows for distributing model information across different layers and having simpler models in lower layers. Graph databases are good examples of this approach. The data model is schema-free, nodes can be labelled but are still not strictly typed since labels do not constrain properties and relationships. However, the data access layer works with objects that map to graph nodes and relationships. Objects are in turn strongly typed by their classes so the mapping will not allow arbitrary properties and connections. The same approach can be followed whenever creational and manipulation tools constrain models.

VI. Conclusion

In this position paper, we have presented our position that models should be kept simple if possible. We have emphasized the importance of using minimalistic models that only contain what is strictly necessary for satisfying the requirements. Of course, the complexity of the model depends on the actual tasks and sometimes complex models cannot be avoided. The point is to aspire to simple models. If requirements change, further details may be added to the model. This is usually easier than dealing with a model that is highly complex from the beginning. Agile software development also shares this point of view. In fact, our position is a generalization of eXtreme Programming (XP), which includes the *"You are not gonna need it" (YAGNI)* and *"do the simplest thing that could possibly work" (DTSTTCPW)* principles [29].

The success of lightweight models have been presented with three technology trends. The first one has explained the limits of general-purpose modeling and how the inherently simpler and more limited domain-specific languages have achieved a higher increase in productivity. After this, we have shown the problems that occur with the strictly structured relational data model and suggested using the simpler and less formalized property graph data model. The last example has discussed Web Services. We have explained how the more limited RESTful approach has proven to be more successful than SOAP Web Services. After these technology trends, we have provided some guidelines on how to use simple models. We believe that this work will help software engineers that work with MDSE to design higher quality and more agile software products.

References

[1] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, Morgan and Claypool Publishers, 2012.
[2] M. Masse, *Rest API Design Rulebook*, O'Reilly, 2011.
[3] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Scalability: The holy grail of model driven engineering," in *Proc. First International Workshop on Challenges in Model Driven Software Engineering, Organized in Conjunction with MoDELS'08*, pp. 1-61, 2008.
[4] L. Burgeño, J. Troya, M. Wimmer, and A. Vallecillo, "On the concurrent execution of model transformations with Linda," in *Proc. the 1st International Workshop on Big MDE*, no. 3, 2013.
[5] L. Layman, L. Williams, and L. Cunningham, "Motivations and measurements in an agile case study," in *Proc. 2004 Workshop on Quantitative Techniques for Software Agile Process*, vol. 52, no. 1, 2004.

[6] S. Kelly and J. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley - IEEE Computer Society Publications, 2008.
[7] M. Fowler, *Domain-Specific Languages*, Addison-Wesley, 2010.
[8] G. Kövesdán, M. Asztalos, and L. Lengyel, "A classification of domain-specific language intents," *International Journal of Modeling and Optimization*, vol. 205, 2012.
[9] J. Partner, A. Vukotic, and N. Watt, *Neo4j in Action*, Manning Publications, 2014.
[10] Object Management Group. (2013). OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/PDF/.
[11] M. Fowler, *UML Distilled, a Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 2003.
[12] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language Reference Manual*, Addison-Wesley Professional, 2004.
[13] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley Professional, 2005.
[14] O. Pastor and J. C. Molina, *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*, Springer, 2007.
[15] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley Professional, 2002.
[16] M. Guttman, *Real-Life MDA: Solving Business Problems with Model Driven Architecture, Morgan Kauffmann*, The MK/OMG Press, 2006.
[17] R. N. Fidalgo, E. M. Souza, S. España, J. B. Castro, and O. Pastor, "EERMM: A metamodel for the enhanced entity-relationship model," in *Proc. the 31st international conference on Conceptual Modeling*, Heidelberg, Germany, pp. 515-524, 2012.
[18] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*, published by O'Reilly Media, 2013.
[19] M. Hunger, *Good Relationships – The Spring Data Neo4j Guidebook*, Neo Technology, 2012.
[20] C. Vickner, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkis, " A comparison of a graph database and a relational database: a data provenance perspective," in *Proc. 48th Annual Southeast Regional Conference*, New York, no. 42, 2010.
[21] K. Barmpis and D. S. Kolovos, "Comparative analysis of data persistence technologies for large-scale models," in *Proc. 2012 Extreme Modeling Workshop*, New York, pp. 33-38, 2012.
[22] C. C. Ribeiro, C. D. Ribeiro, and R. S. G. Lanzelotte, "Query optimization in distributed relational databases," *Journal of Heuristics*, vol. 3, no. 1, 1997.
[23] K. Scribner and M. Stiver, *Understanding SOAP: Simple Object Access Protocol*, Publisher by Sams Indianapolis, 2000.
[24] K. Allen, *WSDL 100 Success Secrets Essentials of Understanding and Applying Web Services Description Language - The XML based protocol for information exchange in decentralized and distributed environments*, Emereo Publishing, 2008.
[25] R. T. Fielding, "Chapter 5, Representational State Transfer (REST)," in *Architectural Styles and the Design of Network-based Software Architectures*, Doctoral Dissertation, University of California, Irvine.
[26] M. Hadley. (2009). Web application description language. W3C Member Submission. [Online]. Available: http://www.w3.org/Submission/wadl/.
[27] T. O'Reilly. (2003). REST vs. SOAP at Amazon blog entry. [Online]. Available: http://www.oreillynet.com/pub/wlg/3005.
[28] V. Amaral, C. Hardebolle, G. Karsai, L. Lengyel, and T. Levendovszky, "Recent advances in multi-paradigm modeling," in *Models in Software Engineering*, Springer Berlin Heidelberg, 2010, pp. 220-24.
[29] G. Succi and M. Marchesi, *Extreme Programming Examined*, Addison-Wesley, 2001.

Gábor Kövesdán earned his MSc degree in computer engineering in 2013. Currently, he is a PhD student at the Department of Automation and Applied Informatics of the Budapest University of Technology and Economics and his area of interest is model processing, domain-specific languages and language parsing.

Márk Asztalos received his PhD in 2013. He is an assistant lecturer in the Department of Automation and Applied Informatics at the Budapest University of Technology and Economics and his research interests include model-driven software development, verification of graph rewriting-based model transformations, and development of domain-specific languages.

He is a member of the visual modeling and software design research group and a developer of the met modeling and model transformation software toolkit visual modeling and transformation system (VMTS).

László Lengyel received his PhD in 2006. He is an associate professor and fellow in the Department of Automation and Applied Informatics at the Budapest University of Technology and Economics. His various research fields focus on software modelling, metamodeling, graph rewriting-based software model transformation, model-driven development, constraint validation, validated model transformation and aspect-oriented techniques. He heads the Visual Modeling and Software Design research group which currently includes 6 researchers and 5 PhD students.

He has published several papers at international conferences, as well as numerous journal articles on topics related to software engineering, software modelling and model processing. The most important milestones in his professional career include, but are not limited to: the Bolyai János professorship (2006-2010), the Siemens Excellence Award (2008), and being chosen as the recipient of the NJSZT Kemény János-award (2012).