An Architectural Proposal for a Model-Driven Environment to Support Lightweight Formal Software Engineering

Elias Adriano Nogueira da Silva[1], **Valdemar Vicente Graciano Neto**[1,2]

[1]Instituto de Ciências Matemáticas e Computação (ICMC)
Universidade de São Paulo - USP
Avenida Trabalhador São-carlense, 400 - Centro
CEP: 13566-590 - São Carlos - SP - Brazil

[2]Instituto de Informática – Universidade Federal de Goiás (UFG)
Alameda Palmeiras, Quadra D, Câmpus Samambaia
P.O Box 131 - CEP 74001-970 - Goiânia - GO - Brazil

Abstract. *Formal methods (FM) are recognized by their characteristics of increasing software quality through the use for formal specifications. On the other hand, Model-Driven Engineering (MDE) is a software development approach that transforms abstract models into software code among others artifacts. However, FM faces difficulties of large-scale adoption in industry due to difficulties developers face when deal with inherent complexity of FM. In this direction, MDE can raise the level of abstraction for FM. Models can be used to increase the level of abstraction in formal specifications, shifting formal notation comprehension to models realizing. An association of FM and MDE creates a lightweight model-driven formal method and improves communication among developers. Such approach provides communication alternatives for stakeholders, and automates verification of formal models, minimizing time and costs. However, tools, approaches, and methods to integrate MDE and FM have not emerged. Thus, as an effort towards a tool support for integrating both paradigms, this paper presents an architectural proposal for MODEFS, a lightweight MOdel-Driven Environment for Formal Specification.*

1. Introduction

Formal Methods (FM) are being increasingly widespread in Software Engineering [Nakajima 1999]. They enhance safety, reliability, and quality of software systems. FM provide formal software specifications, and a set of tools to validate them. However, understand those specificiations is still a complex and time consuming task that can make it difficult to be largely adopted by industry. Model Finders as Z3[1], Model Provers as Darwin [Baumgartner et al. 2004], and Model Checkers as NuSVM [Cimatti et al. 1999] are samples of well-established tools that can be used in a software development process that adopts FM.

On the other hand, Model-Driven Engineering (MDE) is a development approach that relies on high level models to increase the level of abstraction in software production. Therefore, MDE can be associated to FM. These models can raise abstraction in formal specification representations, and automatically transform them into code. Additionally,

[1]`http://rise4fun.com/z3/tutorial`

traditional FM tools for formal specification and formal validation can be associated to this process to create a lightweight model-driven formal method. However, an integrated tool suite still lacks to support such association.

This paper presents MODEFS, a lightweight MOdel-Driven Environment for Formal Specification which composes and integrates FM tools, methods, repositories, and Domain-Specific Languages to create an environment for automatic generation of software based on MDE and FM. The main contribution of this paper is to provide a public architectural proposal of such environment, highlighting its components and how they interoperate. A specific viewpoint is adopted to provide evidences to our findings. Additionally, we built a prototype and a proof of concept implemented as an Eclipse[2] plug-in.

The paper is structured as follows: Section 2 brings background; Section 3 presents the architecture; Section 4 presents the architecture and design of the Lexical, Syntactic and Semantic Analyser and of a Model Finder; Section 5 discusses related work; Section 6 presents conclusions and future works; and Section 7 presents some Acknowledgements.

2. Model-Driven Engineering and Formal Methods

Software development techniques have been improved. However, concerns about systems development, modeling, reuse, productivity, maintenance, documentation, validation, optimization, portability and interoperability are still recurrent. MDE aims at solving these problems [Kleppe et al. 2003, da Silva et al. 2013]. It shifts focus of modern development methodologies from implementation technology to a modeling perspective [Costa et al. 2010].

In MDE, models are first-class citizen [Sendall and Kozaczynski 2003], and transformation mechanisms are used to generate code from abstract models. A model is an abstract description or specification of a system. It is usually represented as a combination of graphical (Domain-Specific Modeling Languages-DSML) and textual elements (Domain-Specific Languages-DSL). Traditionally, models are built in the design step, or even in the earlier stages of the software development life cycle. However, as development proceeds, models become inconsistent and lose their value due to the fact that changes are often made directly in code. Moreover, to create and maintain updated documentation, manual tasks are usually not much appreciated by developers. Thus, models quickly become inconsistent and incapable of representing the system reality [Lucrédio 2009].

Domain-Specific Languages (DSL) are one of the perspectives of MDE implementation. A DSL is a small language, usually declarative, focused on a particular problem/domain [Deursen et al. 2000]. DSL are quite popular and traditional. Remarkable examples include APT language for numerical control (1957-58), and the most famous BNF, or *Backus-Naur Form* language for specifying grammars (1959). Since then, several languages have been proposed, developed and used, what makes the literature in this area very rich [Deursen et al. 2000, Mernik et al. 2005, da Silva et al. 2013]. A DSL can be textual (DSL - allowing to specify programs) or visual (DSML - allowing to specify diagrams). Language definition usually requires a metamodel. A metamodel captures the main concepts of a specific domain.

[2]http://www.eclipse.org/

On the other hand, FM are recognized as a solution for problems in software engineering concerning quality assurance, traceability, correctness, consistency, and others [da Silva et al. 2013]. They have been used to construct models without ambiguity and inconsistencies [Valles-Barajas 2012]. Adoption of FM requires lightweight process [Agerholm and Larsen 1999, Breen 2005]. Software engineering environments should support FM with quite simple mathematical notations to support, for example, requirements specification [Breen 2005].

Model Finders are a class of software used in FM approaches. They verify if a set of logical statements is consistent, i.e, if they are not contradictory between each other. They can be useful, for example, to check consistency in requirements when they are specified in a formal notation. When statements are logically consistent, they are called satisfiable or SAT. Z3 [De Moura and Bjørner 2008] is an example of Model Finder developed by Microsoft Research in USA (An on-line version is available[3]). Essentially, a Model Finder can work under different theories. Z3 is based on satisfiability module theory (SMT). Solvers for SMTs are concerned with the problem of determining the satisfiability of a set of formulas in some first order theory [Reynolds et al. 2013]. Others remarkable notations in FM include Model Checkers and Model Provers. 'Model Prover' is a term used for tools that add theorem proving techniques to the model checking procedures. 'Model Checking' usually assumes that such a proof is done automatically [Berezin 2015]. A model proving usually involves a model checking. A model checking can be performed by a model finder, which finds a combination of values assigned to propositions that make the conjunction of them true. Thus, the model is checked when a model finder finds a satisfiable model.

FM and *Model-Driven* do not often appear in the same sentence [Graciano Neto 2014]. MDE could be applied to FM and vice versa [Gargantini et al. 2010], but there are not many tools that implement these ideas, what can be critical since the adoption of a technology uses to depend on available tools [Hutchinson et al. 2011]. Therefore, the subject should be better explored. Software are represented as models, but there is no guarantees about its correctness, conformance, quality or rigor [Graciano Neto 2014]. On the other hand, applying formal specification and validation is worth since FM allow a software engineer to create consistent, and unambiguous specification. However, FM are not enoughly abstract as the models used in software engineering industry. Thus, an association among these concepts and an associated tool support can bridge a gap on both sides, exploring their inherent complementary weaknesses.

3. A lightweight MOdel-Driven Environment for Formal Specification (MODEFS)

MODEFS is a lightweight environment proposal to support formal specification and validation using models in high level of abstraction. The intention is bridging the gap between formal specifications and code through model transformations. Figure 1 presents the MODEFS architecture under a specific viewpoint, depicting the structural (development) view proposed by Kruchten [Kruchten 1995].

[3]http://z3.codeplex.com/

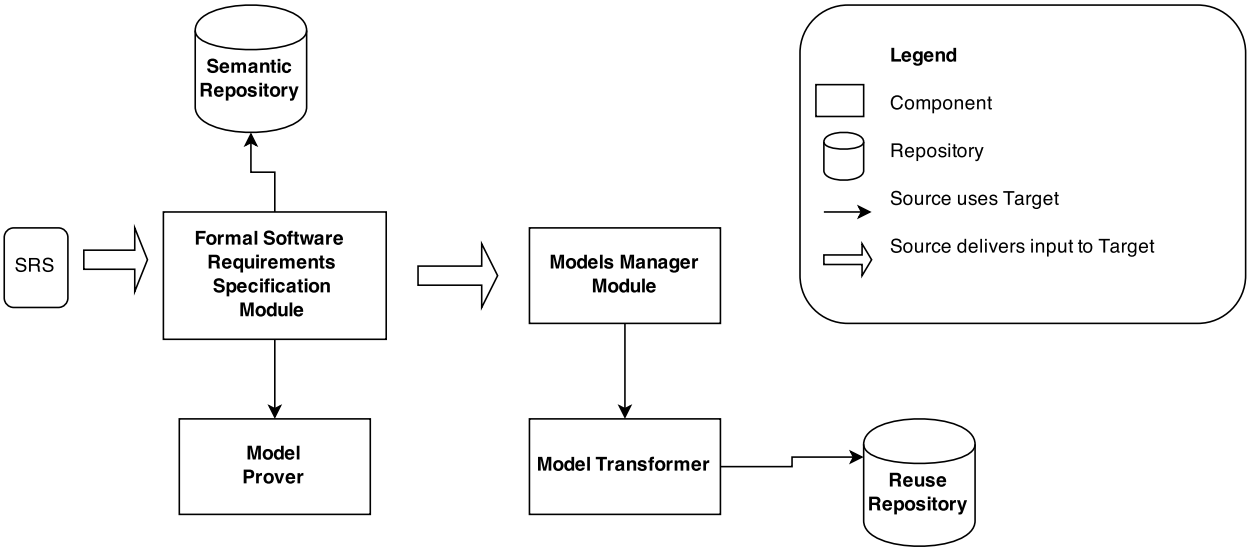

Figure 1. MODEFS: structural view.

The proposed architecture is composed by six modules: **Semantic Repository**, **Reuse Repository**, **Formal Software Requirements Specification (FSRS) Module**, **Models Manager Module**, **Model Prover**, and **Model Transformer**, as depicted in Figure 1. This figure shows that the component located in left side of the arrow uses services provided by that located in the right side of the arrow. Additionally, bold arrows indicates that the element at left side is/delivers an input for the another located in right side of the arrow.

Two repositories are used. Semantic Repository stores language structures to support the automatic identification of requirements. The idea is converting requirements to logic propositions and then validate these requirenments by a Model Prover, such as Z3. Structured and delimited portuguese can be used to specify requirements. An electronic dictionary as WordNet [Gomes et al. 2013] automatically classifies words according to its potential grammar class and subsequently identifying potential logic prepositions structured in a form *Subject Verb Predicate*.

FSRS module uses services provided by the repositories. It is responsible for supporting the software requirements specification. An environment is provided to specify software requirements in a structured way. After specification, services are provided by the Semantic Repository to classify identified words. Those services make it possible to identify phrases that are offered to users in such a way they could confirm if it is really a potential logic preposition that will compose the formal specification.

Model Prover module embodies two integrated tools: a Model Finder (such as Z3) and a Model Checker (such as NuSMV). Model Finder solves the satisfability problem to assure the set of propositions have logic values assigned to them in a way that the conjunction between them are true. Model Checker is responsible to check consistency between those logic prepositions elicited. Consistency, ambiguity, and redundancy are automatically verified using a Model Checker.

Components will be used according to a restrict domain. For example, if a soft-

ware is being developed to a home security system, words as *alarm*, and *invasion* could be associated to COTS (Commercial-of-the-shelf) components that could be composed in a transformation process to generate a software product. Thus, the generated software has a component-based architecture.

Models Manager Module manages models generated by Formal Software Requirements Specification Module, as the formal software requirements specification and other models derived from that one in a more concrete level. And finally, Model Transformer module holds a model transformer component which is used to automatically generate software from formal models.

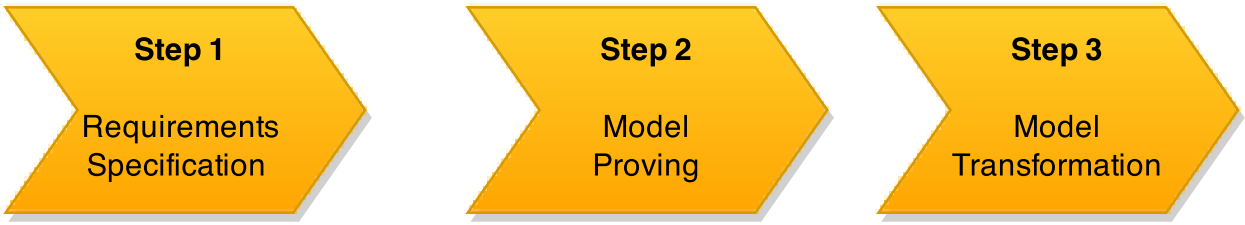

Figure 2. Process proposed to develop software using MODEFS.

A process is proposed to use the tool according to its main modules, as depicted in Figure 2:

1. **Step 1**: Software Requirements Specification (SRS) with textual requirements specified in a structured portuguese was used as input to FSRS module. This module automatically converts those textual requirements in logic propositions written in an specific formal notation through the use of Semantic Repository. Such logic propositions are validated about their consistency with the model finder;
2. **Step 2**: FSRS module originates a formal model as output if requirements are compatible among themselves. Such a model is delivered as input for Model Manager module;
3. **Step 3**: Model Manager submits such model to a model transformer. Model transformer converts that formal model in software properly using Reuse Repository to add COTS elements to the structure.

4. Proof of Concept for a Module of the Architecture

A Model Finder was implemented as a proof of concept. It is a part of the Model Prover module, and integrates the MODEFS environment. The Model Finder engineering was divided in two phases: analyser engineering (the component responsible for lexical, syntactic and semantic analysis), and the SMT solver component engineering.

4.1. A Model Prover Component

MDE techniques and tool were used to support the Model Prover engineering. Xtext[4] was used as an MDE tool to automatically generate an analyser for our model finder. Xtext is a model-driven framework for developing programming languages and DSL. We also implemented a semantic checker.

[4]http://www.eclipse.org/Xtext/

4.2. SMT Solver Component

A Model Finder was implemented as part of the Model Prover to provide Model Checking functionalities. Our Model Finder works under SMT theory.

SMT Solver follows the classic Tableau approach for satisfiability checking [Keller and Heymans 2008]. A Tableau can be represented as a tree data structure. The root stores a conjunction of axioms that have their satisfiability tested. There are specific rules of expanding the tree. In our Tableau, the axioms are expressed in First Order Logic and is restricted to logic formulas. We do not solve mathematical problems as Z3 does. A set of deduction rules is used to construct the tableau. During the solution of Tableau, priority is given to the solution of axioms with "implies" and "and" operations. "Or" operations are solved as late as possible since they generate the opening of Tableau branches and branches can open indefinitely. If the opening is unavoidable, it is done.

4.3. Example

This section brings a practical illustration of how MODEFS is supposed to be used and applied using the process we detailed in the last section. Our practical example illustrates some requirements and logic propositions related to a Home Security System.

Following the process proposed to use MODEFS, a meeting with the customer happens. Requirements elicitation activities are performed, accomplishing Step 1. An SRS is produced as the output of this activity. After this step, the Formal Software Requirements Specification (FSRS) module is used to manage the requirements that were collected. A model based on IEEE 830 standard [IEEE 1998] is used to document some requirements. An excerpt of such a document is presented in Table 1.

An alarm system works as follows: if it is turned on and a invasion happens, then triggers the alarm; if it is turned on and a invasion does not happen, no alarm. If it is off, regardless of there is or not an invasion, alarm is not triggered.

Table 1. An excerpt of a Software Requirements Speficication.

In Step 2, using knowledge extraction techniques, FSRS module extract logic propositions from the SRS document supported by the Semantic Repository. Such repository is capable of identifying potential grammar classes, converting text in logic propositions. Propositions are aligned with the aforementioned format *Subject Verb Predicate*. Predicate is not mandatory:

- p: alarm triggers.
- q: alarm is on.
- r: invasion happens.

One possible logic formula that could be extracted from that excerpt is the illustrated in following formula:

$$p \leftrightarrow q \wedge r \tag{1}$$

The user (software engineer) is required to choose between conditional or biconditional operators since the semantic influences the final result. In this case, biconditional

is ideal since the alarm must not be fired when the alarm is turned off or when a invasion does not happen, as illustrated in Table 2.

p	q	r	$q \wedge r$	$p \leftrightarrow q \wedge r$
T	T	T	T	T
T	T	F	F	F
T	F	T	F	F
T	F	F	F	F
F	T	T	T	F
F	T	F	F	T
F	F	T	F	T
F	F	F	F	T

Table 2. Truth-table for the presented formula.

Table 2 have *true* values represented by character *T* and *false* values represented by character *F*. Only the first, sixth, seventh, and eighth lines are true. This have an specific meaning. It can be read in the following way: *the assignments of values to propositions are true when*: 1) alarm rings if it is on and invasion happens (p = T, q = T, r = T); 2) alarm does not ring if it is on but an invasion does not happen (p = F, q = T, r = F); 3) alarm does not ring if it is off (independently if there is an invasion or not - seventh and eighth lines: p = F, q = F, r = T or F). This exactly represents the behavior we expect for situations where the alarm is on and a invasion occurs. Other situations are consequences.

Logical treatment is done in the FSRSM. After that, the model is used as input to the Model Prover where the consistency among propositions is checked. In our example, we have only one proposition, but the work is designed to support a lot of them. Consistency checking accomplishes Step 2.

Following the process, Model Manager Module uses the Model Transformer to transform the model with other modules in a functional software, starting Step 3. Components available in the Reuse Repository are used to compose the software product architecture. The architecture is predefined (as components for an usual home security system), and the components registered for this kind of software are available in the repository. Thus, the transformer can compose the software with quality assurance, since the input model was validated and the software components are supposed to be tested in other projects of the portfolio.

After that, the software product is complete and the customer can validate. The cycle can be repeated again if requirements have been eventually misunderstood.

A first step towards the complete environment implementation was done. A part of the Model Prover was implemented, and part of the generation used MDE paradigm. Requirements granularity and conventions to document requirements to convert them in logic propositions are not subject of this paper.

5. Related Work

Other remarkable initiatives to associate MDE and FM are available in literature. We highlight two main tool proposals: Perfect Developer and Echo.

Perfect Developer[5] is a software development tool for developing formal specifications and transforming them into code. The tool uses advanced automated reasoning to discharge proof obligations without user intervention. Advanced mathematical knowledge is not a pre-requisite, which means that any developer fluent in an object-oriented language such as Java or C++ can be able to learn the notation. Perfect Developer is similar to our proposal regarding the use of MDE in association with FM. However, in their proposal, users must specify requirements using a DSL called Perfect. In our proposal, we adopt structured portuguese, which is more abstract. Futhermore, we present flexible process and architecture that any developer can use to build its own approach that can be more aligned with its needs. The elements we presented are not rigid and unstead using structured portuguese the developer could use, for example, english.

In parallel, Echo[6] is a tool for model transformation based on the Alloy model finder. It is also built over the Eclipse Modeling Framework (EMF). Echo is meta-model independent, being able to process any meta-model specified in ECore and its respective instances in XMI. Additional constraints, as well as operations, are specified by embedding OCL in annotations, as prescribed by EMF. Inter-model consistency is specified by the QVT[7] Relations (QVT-R) transformation language. Echo works in a level similar to Perfect Developer. In their tool, requirements must be directly specified in Alloy, what is not trivial. Our proposal involves natural language, a human-readable abstraction that becomes our proposal easier to use.

Furthermore, Miller et al. [Miller et al. 2010] also claims for a Model-Based approach for formal software engineering. They describe a translator framework developed by Rockwell Collins and the University of Minnesota that allows them to automatically translate from commercial modeling languages to a variety of model checkers and theorem provers, validating the software code. Analogously to other proposals, they do not work on automatic transformation of requirements to code, but from code to validated code. Thus, our proposal is broader.

6. Conclusions and Future Work

This article presented an architectural proposal of an environment to support a lightweight formal specification and validation for software based on MDE principles. The main contribution against similar tools is proposing an automatic extraction of formal models from a set of requirements specified in structured portuguese, a set that is not covered by other proposals. Formality is provided by extracting logic propositions from a Software Requirements Specification (SRS), transforming them in a textual formal model using some type of formalism. Thus, formal specification must only be validated. Specification effort remains at natural language level.

An Eclipse-based plugin was implemented as a prototype to perform model checking of such a formal model with a SRS as a proof of concept. Requirements Engineering is supported through the use of a controlled vocabulary to encompass linguistic equivalences to support automatic transformation of structured language to a formal notation. A

[5] http://www.eschertech.com/products/perfect_developer.php
[6] https://github.com/haslab/echo
[7] http://www.omg.org/spec/QVT/

prototype of a Model Prover component was implemented as a proof of concept. Additional refinements are necessary for the plain working of such a Component.

As future works, we envision the use of complementary models such as Software Requirements Specifications, UML class diagrams, use cases, and activity diagrams could be adopted to SRS source model. Additionally, other functionalities specified at this architectural proposal must be implemented to accomplish the whole established goal of providing such an environment to support a lightweight model-driven formal method engineering. Another future work can be increasing the scope of work to encompass other types of logic. After all, propositional logic has limitations in semantics and expressiveness. Other formalisms can be adopted to increase expressiveness as First-Order Logic, Modal Logic, and Temporal Logic.

Indeed, this is a first step towards effectively bringing formal methods (FM) to software engineering tools reality. We believe that our contributions can foster the adoption of FM since it makes it easy to use FM in a software specification and in automatic generation of software from abstract models.

7. Acknowledgments

The authors would like to thank FAPESP (process number 2012/244887-3) and FAPEG (grant number 09/2013) for partially funding this research.

References

Agerholm, S. and Larsen, P. (1999). A lightweight approach to formal methods. In Hutter, D., Stephan, W., Traverso, P., and Ullmann, M., editors, *Applied Formal Methods — FM-Trends 98*, volume 1641 of *LNCS*, pages 168–183. Springer Berlin Heidelberg.

Baumgartner, P., Fuchs, A., and Tinelli, C. (2004). Darwin: A Theorem Prover for the Model Evolution Calculus.

Berezin, S. (2015). Symp: Symbolic model prover. http://www.cs.cmu.edu/ modelcheck/symp.html. Last Access: April 2015.

Breen, M. (2005). Experience of using a lightweight formal specification method for a commercial embedded system product line. *Requirements Engineering*, 10(2):161–172.

Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999). Nusmv: A new symbolic model verifier. In *ICCAV*, pages 495–499, London, UK, UK. Springer-Verlag.

Costa, S. L., Graciano Neto, V. V., Loja, L. F. B., and de Oliveira, J. L. (2010). A Metamodel for Automatic Generation of Enterprise Information Systems. BWMDD '10, pages 45–52, Salvador, BA, Brazil. UFBA.

da Silva, E. A. N., Fortes, R. P. M., and Lucredio, D. (2013). A Model-Driven Approach for Promoting Cloud PaaS Portability. In *CASCON*.

De Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *Proc. of the Theory and Practice of Software*, pages 337–340, Berlin, Heidelberg. Springer-Verlag.

Deursen, V., , Klint, A., and Paul and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.

Gargantini, A., Riccobene, E., and Scandurra, P. (2010). Combining formal methods and mde techniques for model-driven system design and analysis. *International Journal On Advances in Software*, 1:1–18.

Gomes, M. M., Beltrame, W., and Cury, D. (2013). Automatic Construction of Brazilian Portuguese WordNet. In *X ENIAC*, pages 1–10. UFCE.

Graciano Neto, V. V. (2014). Model-Driven Development and Formal Methods: A Literature Review. ENACOMP '14, pages 1–10.

Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. (2011). Empirical assessment of mde in industry. ICSE '11, pages 471–480, New York, NY, USA. ACM.

IEEE (1998). IEEE Recommended Practice for Software Requirements Specifications. Technical report.

Keller, U. and Heymans, S. (2008). The sat-tableau calculus. In *Proceedings of the 21st International Workshop on Description Logics (DL2008), Dresden, Germany*. CEUR-WS.org.

Kleppe, A., Jos, W., and Wim, B. (2003). MDA Explained, The Model-Driven Architecture: Practice and Promise. Addison-Wesley.

Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50.

Lucrédio, D. (2009). Uma Abordagem Orientada a Modelos para Reutilização de Software. Tese de Doutorado.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344.

Miller, S. P., Whalen, M. W., and Cofer, D. D. (2010). Software model checking takes off. *Communications of ACM*, 53(2):58–64.

Nakajima, S. (1999). Using algebraic specification techniques in development of object-oriented frameworks. In Wing, J., Woodcock, J., and Davies, J., editors, *FM'99 — Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1664–1683. Springer Berlin Heidelberg.

Reynolds, A., Tinelli, C., Goel, A., Krstic, S., Deters, M., and Barrett, C. (2013). Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *CADE-24*, LNCS, pages 377–391. Springer.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45.

Valles-Barajas, F. (2012). Using Lightweight Formal Methods to Model Class and Object Diagrams. *Computer Science and Information Systems*, 9(1):411–429.