MDEO-Cloud: A Modular Cloud-Based Platform for Model-Driven Optimization

Niklas Krieger
University of Stuttgart
Stuttgart, Germany
niklas.krieger@iste.uni-stuttgart.de

Sandro Speth
Technical University of Munich
Munich, Germany
sandro.speth@tum.de

Steffen Becker
University of Stuttgart
Stuttgart, Germany
steffen.becker@iste.uni-stuttgart.de

Steffen Zschaler
King's College London
London, UK
szschaler@acm.org

Abstract

Model-driven optimization (MDO) uses domain-specific modeling languages (DSMLs) to describe and solve optimization problems, aiming to reduce complexity for domain experts who lack optimization expertise. However, existing research prototypes remain fragmented and tightly coupled to the Eclipse ecosystem, making them difficult to access and reproduce. We present MDEO-Cloud, a cloud-based platform for model-driven optimization that provides a unified, web-accessible environment for collaborative research and development. Built on a plugin-based architecture, the platform supports extensibility through language and contribution plugins, enabling seamless integration of new optimization approaches. We demonstrate the platform's capabilities through a scalable execution engine that features federated distributed computing and present a family of DSMLs supporting blended graphical-textual modeling, which we evaluate through a user study. We hope that MDEO-Cloud can serve as a common foundation for the MDO research community, facilitating collaboration and accelerating progress in the field.

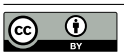
CCS Concepts

• **Software and its engineering → Domain specific languages**; **Integrated and visual development environments**; • **Mathematics of computing → Combinatorial optimization**; *Graph algorithms.*

Keywords

Model-Driven Optimization, Domain-Specific Modeling Languages, Cloud-Based Platform

ACM Reference Format:
Niklas Krieger, Sandro Speth, Steffen Becker, and Steffen Zschaler. 2026. MDEO-Cloud: A Modular Cloud-Based Platform for Model-Driven Optimization. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3837062.3838866



MODELS Companion 2026, Málaga, Spain

ACM ISBN 979-8-4007-2903-4/2026/10
https://doi.org/10.1145/3837062.3838866

1 Introduction

Model-driven optimization (MDO) [3, 14] aims to reduce the complexity of specifying and solving optimization problems for domain experts who lack optimization expertise. A typical approach [1, 6, 10] is to capture the search space in the metamodel of a domain-specific modeling language (DSML) and use model-to-model transformations (often realized as graph transformations) to specify the decisions that can be made to find a feasible and optimal solution. Objective functions and constraints are then specified using standard constraint languages such as OCL [22] or model queries in suitable programming languages.

Many research prototypes for MDO have been developed in the Eclipse IDE, benefiting from its rich plugin ecosystem and the availability of mature modeling frameworks such as EMF [24]. However, this has increasingly become a barrier to adoption, and the modeling community has been moving toward more web-based approaches over the last few years. While there are examples of web-based MDO tools (e.g., Refinery [20]), these remain rare and do not provide a unified platform for comparative research (e.g., [13]) and the collaborative development of new MDO approaches.

We address this gap by presenting MDEO-Cloud, a cloud-based platform for model-driven optimization designed for extensibility from the start, enabling researchers to easily integrate new optimization approaches and DSMLs. Inspired by the web-based plugin architecture of the MDENet Education platform [27], MDEO-Cloud plugins can be hosted on any web server and added to individual optimization projects at runtime, without requiring any changes to the platform itself. We demonstrate the platform with an initial set of plugins that provide basic modeling functionality (fully supporting blended modelling [7]) and a scalable execution engine that supports federated distributed computing, using an evolutionary algorithm as the solver.

We hope that MDEO-Cloud can serve as a common foundation for the MDO research community, facilitating collaboration and accelerating progress in the field, and encourage others to use it and contribute to its development. The platform is open-source and available at https://github.com/mde-optimiser/mdeo-cloud/, with documentation at https://mde-optimiser.github.io/mdeo-cloud/, a demonstration video at https://youtu.be/Oh9z-NVjO4Y, and a replication package for our evaluation on Zenodo [17].

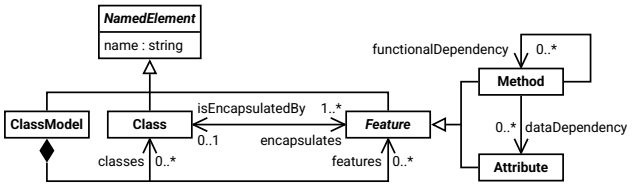


Figure 1: The CRA metamodel (from [11]).

2 Running Example: CRA

We introduce MDEO-Cloud using the Class Responsibility Assignment (CRA) problem, a software engineering case study [11] that is a widely used MDO benchmark. A problem instance is a responsibility dependency graph (RDG) of `Features` and their dependencies, captured as a metamodel (Figure 1): `Attributes` have no dependencies, while `Methods` can depend on both `Attributes` and other `Methods`. The goal is to assign `Features` to `Classes` (creating classes as needed) so that the resulting `ClassModel` is of high quality, expressing class creation and feature assignment as graph transformation rules (one is shown in Figure 2). Quality is measured by the CRA-Index [11], which maximizes cohesion of `Classes` while minimizing coupling between them, yielding a single-objective problem.

3 A Configurable, Web-based Modeling Platform for Optimization Problems

Our platform is structured around five conceptual areas: project organization, user management, plugin-based extension, file storage, and execution. Users are authenticated actors within the system, identified by their username, with permissions managing their rights. Projects are the platform's central organizational units, with their content defined by a file tree. For each file, the extension indicates which language it belongs to and, therefore, the responsible plugin. Associated file metadata allows editors to store required information hidden from the primary textual syntax, e.g., layout information for graphical editors.

Architecturally, the underlying platform consists of a web-based frontend connected to a cloud backend that provides REST and WebSocket APIs via an API gateway, as shown in Figure 2. The core backend, which handles project and plugin management, is connected to a set of plugin services.

Plugins. Throughout the platform, we utilize a plugin-based architecture to enable extensibility, particularly for other MDO approaches. Plugins are configured on a per-project level, specifying which languages are available within a project. Each plugin consists of a set of *language plugins*, and a set of *contribution plugins*. A *language* plugin provides support for a single modeling language, and defines its concrete textual and/or graphical syntax. To allow for further flexibility, languages can define extension points, which can be utilized by *contribution plugins*. Each plugin is defined via its URL pointing to the plugin service, from which the *plugin manifest* is fetched by the main backend. Subpaths are used for various plugin functionalities, including serving frontend-required static files and triggering executions. Executions typically depend on data defined by files from other plugins. Plugin services obtain such *file data*, defined by the respective file and a *key*, by requesting it from the core backend, which forwards the request to the responsible plugin service and handles caching. The core backend tracks dependencies between files and their computed file data, using this information to invalidate the cache when a dependency changes.

Execution. Executions model a task run of a specific file, and are handled by the responsible plugin service. During execution, a task moves through the states `SUBMITTED`, `INITIALIZING`, `RUNNING`, and finally `COMPLETED` or `FAILED`; users can cancel an execution in any non-terminal state, transitioning it to `CANCELLED`. When the execution either completes or fails, a Markdown report is generated. Furthermore, depending on the type of execution, a file tree similar to that of the project can be generated and can contain additional file-based results. Executions capture both their start and finish timestamps and can thus measure the execution duration. Additionally, a progress message can be continuously updated to indicate that execution is still in progress.

Frontend. Figure 2 shows the main layout of the web frontend. It takes inspiration from both editors and IDEs like VS Code, as well as web-based modeling platforms like Dandelion [19], and consists of four major components: On the left edge, the *activity bar* contains two groups of icon buttons: the top group controls the *sidebar*, and the bottom group provides access to settings and account management. The closeable and resizable *sidebar* is located to the right, with Figure 2 showing the *file tree* of the project. Other supported sidebars include project management, search, and execution. The *main editor area* fills the remainder of the viewport, consisting of the *tab bar* at the top and the *editor* below: The *tab bar* shows the currently opened tabs, as well as *file actions*, e.g., used for starting executions based on the active tab. Depending on the language of the file backing the active tab, either a graphical, textual, or blended graphical-textual editor is provided to the user, with Figure 2 showing the blended editor for the *model transformation* language. While textual editors are provided by Langium, graphical editors are implemented using Eclipse GLSP with a highly customized frontend, including a custom plugin-customizable toolbox. Both views are automatically synced while preserving representational information in each view.

4 Initial Instantiation: Porting MDEOptimiser

We reimplement MDEOptimiser [6] using our foundational platform, implemented as a set of language and contribution plugins, and showcase its functionality using the CRA example. Instead of reusing existing languages, we design a new family of languages to achieve a high level of syntactic consistency and improve usability, with a particular focus on blended graphical-textual modeling. At the core of our language family lies the *metamodel* language, which replaces Ecore and is used to define the problem and solution space of an optimization problem. Consisting of classes, enums, and associations, it is semantically highly aligned with EMOF. It supports a blended graphical-textual editor, as shown in Figure 2, with the graphical syntax using UML class diagrams, while the textual syntax draws heavy inspiration from both PlantUML and programming languages like Java.

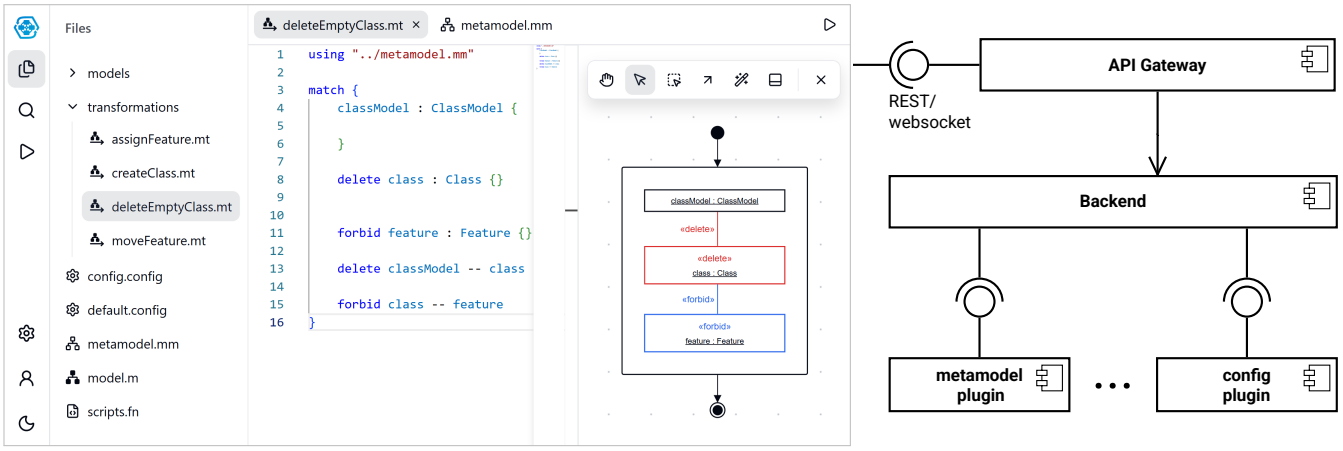


Figure 2: MDEO-Cloud platform (left) with an example model transformation, and the backend architecture (right).

The *model* language defines metamodel-conforming models that serve both as the initial input defining the concrete optimization problem to be solved and as the population individuals that the evolutionary algorithm maintains and evolves during optimization. Similar to the metamodel language, we utilize a blended graphical-textual editor, using UML object diagrams for graphical representation. The *expression* language serves as a shared foundation used by two higher-level languages to provide a common, statically typed expression and statement syntax.

The *model transformation* language replaces Eclipse Henshin [25] and defines metamodel-conforming mutation operators that explore the solution space as endogenous [8] model transformation; it uses the *expression* language for conditions and computed property values. Using a story-driven modeling [21] approach, it consists of an outer control flow layer with embedded graph transformation rules, so-called match patterns: Match patterns are defined similarly to Eclipse Henshin rules, with support for both negative (NAC) and positive application conditions (PAC), as well as for the left-hand side (LHS) and right-hand side (RHS) graphs. For the CRA problem, we use four model transformations in total, one of which we show in Figure 2, consisting of a single match step with a NAC. Textually, we utilize the same syntax as the *model* language, with additional keywords which indicate to which graphs instances and links belong: `create` for RHS-only elements, `delete` for LHS-only elements, and `forbid` and `require` for NACs and PACs, respectively. Elements present in both LHS and RHS graphs are not annotated to reduce modeling effort. Graphically, as with most existing approaches, we use a unified graph representation in which the same keywords are represented via UML keyword labels. Further, color highlighting improves readability, with green indicating create, red indicating delete, blue indicating forbid, and orange indicating require. The control flow layer consists of a UML activity diagram-like sequence of control flow statements that embed the graph transformations. It supports conditional and iterated execution, both based on conditional matches and expressions. Graphically, similar to story diagrams, it uses the UML activity diagram's concrete syntax, while the textual syntax is strongly inspired by the PlantUML syntax for activity diagrams, as well as by regular programming languages.

Next, the *script* model query and constraint language is used to define both objectives and constraints that are executed to evaluate models during optimization runs; it subsumes the Java/OCL/Xtend-based constraint and objective function definitions of the original MDEO. Based on the common *expression* language, it allows functions to be defined as sequences of executed statements. It is a statically typed, lexically scoped language that primarily focuses on imperative programming concepts, with some object-oriented and functional aspects. Syntactically, it is strongly inspired by Kotlin and provides ways to interact with the model instance similar to OCL and the Epsilon Object Language [15]. For the CRA problem, we define a single objective function that computes the CRA-Index and another objective that ensures all features are assigned to a class.

Finally, the *config* language fully specifies an optimization run. Designed as an extensible configuration language, it aggregates configuration sections via contribution plugins. Our *config-optimization* plugin provides general MDE-based optimization support. It defines a *problem* section, which references the solution-space-defining metamodel and problem-defining model; and a *goal* section, which defines optimization objectives and constraints via referenced *script* functions. Building on top of *config-optimization*, our *config-mdeo* plugin contributes *search*, *solver*, and *runtime* sections. The *search* section defines the search space using search operators. Supported mutation operators include manually defined model transformations and generated consistency-sustaining mutation operators [5, 16]. The *solver* and *runtime* sections allow configuring both the optimization algorithm and its execution characteristics.

5 Simple, Scalable, and Observable Execution

Our optimization execution service is based on the `core` library of the original MDEOptimiser. Ecore metamodels and models are replaced with an implementation for our custom *metamodel* and *model* language, which uses ASM [18] to generate Java classes at runtime. Java, OCL, and Xtend guidance functions are replaced by our custom *script* language, for which we also implement a runtime compiler using ASM, avoiding slow interpretation at runtime. For search operators, we utilize our *model transformation* language

instead of Eclipse Henshin, for which we implement a Gremlin-based [23] runtime which uses an in-memory graph database for model transformation execution. Like the original MDEOptimiser, underlying evolutionary optimization algorithms are provided by the MOEA framework [12].

We integrate horizontal scalability, both to multiple threads and to multiple nodes, while maintaining the semantics of the original implementation. The MOEA framework provides two main ways for parallelization: First, the `DistributedProblem` allows for parallelizing problem evaluation using a Java `ExecutorService`, e.g., to multiple threads. However, for execution on multiple nodes, this results in significant (de)serialization overhead, as models are stored on the central node running the evolutionary algorithm. In contrast, the island model splits the population into smaller subpopulations [9]. Each subpopulation evolves independently, and periodically exchanges solutions via *migration*. While the island-based approach often performs well in practice, it still changes the semantics of the evolutionary algorithm, which we want to avoid.

Instead, we utilize a federated approach, where the solutions are distributed over several nodes, and a central orchestrator handles the evolutionary algorithm. On the orchestrator, solutions consist of a reference to a model on a potentially remote node, a reference to the parent (if one exists), and the constraint and objective values. When starting the optimization, the initial node acts as an orchestrator and fetches all required data (scripts, metamodel, model, model transformations) from the backend. Based on the runtime configuration, it then connects to the required amount of peer nodes, and provisions stateful executors. During the evolutionary algorithm execution, when applying a search operator, the algorithm determines which solution(s) to evolve, and transmits this information to the executors. Each of these execution nodes then uses one or more threads to first generate the new models, followed by applying the guidance functions. Afterward, the results are sent back to the orchestrator, which continues the evolutionary algorithm via the MOEA framework. As surviving solutions are not distributed evenly, rebalancing between nodes is required. At the start of each iteration, the orchestrator decides which solutions need to be relocated to other nodes; the actual relocation is performed directly between nodes.

Moreover, we improve optimization execution by reducing the number of model transformations that need to be performed. Depending on the evolutionary algorithm, solutions can be kept over several generations. When applying a mutation operator, if the application fails, another random operator is tried, until no further operators can be tried. However, if failure can be determined deterministically, attempting the same operator again does not provide any benefit, and can therefore be skipped. This is highly beneficial, as determining a non-match (e.g., using NACs) is often expensive.

During execution, the platform provides observability via a continually updated message in the frontend, including the number of finished batches and performed model transformations. Further, the generated markdown report provides an overview of the obtained solutions (constraint and objective values), and displays several additional graphs. These show the total execution time per generation, the number of model transformations performed and skipped, and the number of solutions transferred to other nodes. Further, we show how the hypervolume quality indicator develops over time.

6 User Study

To assess the user experience of MDEO-Cloud compared to the original Eclipse-based *MDEOptimiser*, we conducted a controlled user study in which $n = 8$ participants performed an identical model-driven optimization task in both tools. The study pairs a within-subjects controlled experiment, which measures task completion time, with a post hoc survey, which measures perceived usability.

Study Design. We use a *counterbalanced within-subject* design [26]. Eight participants with a strong software-engineering background are randomly assigned to two equally sized groups that differ only in the order of tools; this controls for learning and fatigue effects across the two treatments. Both groups solve the *same* task—a Scrum sprint-planning case study—using both tools, so the modeling experience is directly comparable, while a separate 0–1 knapsack example is used only for training. Each session comprises (i) a conceptual introduction to model-driven evolutionary optimization, (ii) an introduction to the first tool, (iii) a six-part modeling task using the first tool, (iv) the previous two steps repeated with the second tool, and (v) the survey. The task spans the full workflow: defining the metamodel, two model transformations, an instance model, one constraint function, the optimization configuration, and finally running the optimizer and inspecting the results; the remaining transformations and guidance functions were pre-supplied to keep each session within two hours. We operationalize *usability* using the System Usability Scale (SUS) [4] and *efficiency* via task completion time, and expect MDEO-Cloud to improve both over MDEOptimiser. The survey additionally records prior experience across seven MDE-related dimensions, five-point Likert items on perceived effectiveness and helpfulness of blended modeling, four direct tool-comparison items, and eight open-ended questions to interpret the quantitative findings.

Results. MDEO-Cloud raises the mean SUS score from 31.2 to 81.2; on the adjective scale of Bangor et al. [2], this moves the tool from between *worst imaginable* and *poor* to the upper end of *good*. Every participant rated MDEO-Cloud higher, and the mean task completion time dropped from 38 to 28 minutes, with all eight participants completing the second task faster despite an 11.9% learning effect. The Likert items corroborate this: all eight participants preferred MDEO-Cloud and agreed it let them work more effectively, while 7/8 found blended modeling helpful and 6/8 preferred its browser-based experience to Eclipse.

The open-ended answers explain these gains: six of eight participants found MDEO-Cloud both faster and easier to learn, citing autocomplete, a cleaner, more tightly integrated user interface, and a live-synchronized blended graphical-textual editor. MDEOptimiser was most often slowed down by missing autocomplete and general Eclipse/IDE overhead, while the most requested improvement for *both* tools was more workflow guidance (e.g., configuration wizards or templates). Blended modeling and autocomplete were the top features of MDEO-Cloud, plus zero-install web access.

Both objective (task time) and subjective (SUS, preference) measures indicate that MDEO-Cloud substantially improves usability and efficiency over the original MDEOptimiser, with participants attributing the improvement chiefly to blended modeling, autocomplete, and the zero-install web client. The user study involved a

small number of participants and a relatively small task; a more extensive usability evaluation is left to future work.

7 Scalability Evaluation

To evaluate the scalability of our optimization backend, we conduct a controlled experiment to measure the optimization execution time for the CRA case study under varying levels of parallelism. MDEO-Cloud is deployed with three optimizer execution nodes on a workstation running Ubuntu 24.04, equipped with an AMD Ryzen™ 9 9950X3D 16-core processor and 96 GB of DDR5 RAM.

We use five input models of varying size and complexity taken from [11]. For each model, we evaluate twelve configurations obtained by combining 1–3 optimizer nodes with 1–4 threads each, allowing us to assess both intra-node (thread-level) and inter-node (distribution-level) scalability. Execution time, measured internally by MDEO-Cloud, covers only the optimization phase and is averaged over 20 independent runs per configuration.

Increasing the thread count on a single node to four threads already yields a speedup of 2.49–3.28× across the input models. The fastest configuration is three nodes with four threads each, which achieves a speedup of 3.86–4.34× over the baseline. Notably, running three threads on a single node (2.16–2.70×) is consistently more efficient than distributing the same amount of parallelism across three nodes with a single thread each (1.48–2.09×). This can be explained by the communication overhead and the (de)serialization overhead incurred when migrating models between nodes.

8 Conclusion and Outlook

We have presented MDEO-Cloud, a web-based MDO platform whose plugin architecture lets researchers integrate new optimization approaches and DSMLs without modifying the core platform. We demonstrated it with an initial set of plugins providing blended modeling and a scalable execution engine using federated distributed computing with an evolutionary solver, and hope it serves as a common foundation to accelerate future MDO research.

Future work will expand the supported DSMLs and optimization approaches (e.g., MILP-based solvers), add features such as richer visualization of optimization results, and explore more efficient translations of model-based problem specifications into formulations that are efficiently solvable by off-the-shelf solvers.

AI Statement

Part of the implementation of MDEO-Cloud, as well as some of the evaluation support scripts, were generated with the help of GitHub Copilot (using a range of different models).

References

[1] Hani Abdeen, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. 2014. Multi-objective Optimization in Rule-based Design Space Exploration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, 289–300. doi:10.1145/2642937.2643005

[2] Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction* 24, 6 (2008), 574–594.

[3] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. 2017. A survey on search-based model-driven engineering. *Automated Software Engineering* 24, 2 (April 2017), 233–294. doi:10.1007/s10515-017-0215-4

[4] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[5] Alexandru Burdusel, Steffen Zschaler, and Stefan John. 2021. Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering. *Software and Systems Modeling* 20, 6 (Aug. 2021), 1857–1887. doi:10.1007/s10270-021-00914-w

[6] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. 2018. MDEoptimiser: A Search Based Model Engineering Tool. In *Companion Proc. 21st ACM/IEEE Int'l Conf Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18)*. ACM, New York, NY, USA, 12–16. doi:10.1145/3270112.3270130

[7] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. 2019. Blended Modelling – What, Why and How. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 425–430. doi:10.1109/models-c.2019.00068

[8] K. Czarnecki and S. Helsen. 2006. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45, 3 (2006), 621–645.

[9] Ke-Lin Du and M. N. S. Swamy. 2016. *Search and Optimization by Metaheuristics: Techniques and Algorithms Inspired by Nature*. Springer International Publishing, Cham. doi:10.1007/978-3-319-41192-7

[10] Martin Fleck, Javier Troya, and Manuel Wimmer. 2015. Marrying Search-Based Optimization and Model Transformation Technology. In *Proc. 1st North American Search Based Software Engineering Symposium (NasBASE'15)*. Preprint available at http://martin-fleck.github.io/momot/downloads/NasBASE_MOMoT.pdf.

[11] Martin Fleck, Javier Troya Castilla, and Manuel Wimmer. 2016. The class responsibility assignment case. (2016).

[12] David Hadka. 2025. *MOEA Framework: A Free and Open Source Java Framework for Multiobjective Optimization*. Computer software. https://github.com/MOEAFramework/MOEAFramework

[13] Stefan John, Alexandru Burdusel, Robert Bill, Daniel Strüber, Gabriele Taentzer, Steffen Zschaler, and Manuel Wimmer. 2019. Searching for Optimal Models: Comparing Two Encoding Approaches. *The Journal of Object Technology* 18, 3 (2019), 6:1. doi:10.5381/jot.2019.18.3.a6

[14] Stefan John, Jens Kosiol, Leen Lambers, and Gabriele Taentzer. 2023. A graph-based framework for model-driven optimization facilitating impact analysis of mutation operator properties. *Software and Systems Modeling* 22, 4 (Jan. 2023), 1281–1318. doi:10.1007/s10270-022-01078-x

[15] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2006. The Epsilon Object Language (EOL). In *Model Driven Architecture – Foundations and Applications*, Arend Rensink and Jos Warmer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 128–142.

[16] Jens Kosiol, Daniel Strüber, Gabriele Taentzer, and Steffen Zschaler. 2022. Sustaining and improving graduated graph consistency: A static analysis of graph transformations. *Science of Computer Programming* 214 (Feb. 2022), 102729. doi:10.1016/j.scico.2021.102729

[17] Niklas Krieger, Sandro Speth, Steffen Zschaler, and Steffen Becker. 2026. *MDEO-Cloud: A Modular Cloud-Based Platform for Model-Driven Optimization - Replication Package*. doi:10.5281/zenodo.21888711

[18] Romain Lenglet. 2002. ASM: a code manipulation tool to implement adaptable systems. *, Adaptable and extensible component systems* (2002).

[19] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. 2023. Dandelion: A scalable, cloud-based graphical language workbench for industrial low-code development. *Journal of Computer Languages* 76 (2023), 101217. doi:10.1016/j.cola.2023.101217

[20] Kristóf Marussy, Attila Ficsor, Oszkár Semeráth, and Dániel Varró. 2024. Refinery: Graph Solver as a Service: Refinement-based Generation and Analysis of Consistent Models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24)*. ACM, 64–68. doi:10.1145/3639478.3640045

[21] Ulrich Norbisrath, Ruben Jubeh, and Albert Zündorf. 2013. *Story Driven Modeling*. CreateSpace Independent Publishing Platform.

[22] Object Management Group. 2014. Object Constraint Language (OCL). online: https://www.omg.org/spec/OCL/.

[23] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (Pittsburgh, PA, USA) *(DBPL 2015)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/2815072.2815073

[24] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *Eclipse Modeling Framework, 2nd Edition*. Pearson Education.

[25] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. 2017. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In *Proc. 10th Int'l Conf on Graph Transformations (ICGT'17)*, Juan de Lara and Detlef Plump (Eds.). Springer, 196–208.

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2024. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg. 73–83 pages. doi:10.1007/978-3-662-69306-3

[27] Steffen Zschaler, Will Barnett, Artur Boronat, Antonio Garcia-Dominguez, and Dimitris Kolovos. 2025. The MDENet education platform: zero-install directed activities for learning MDE. *Software and Systems Modeling* 25, 1 (June 2025), 287–313. doi:10.1007/s10270-025-01292-3