Cross-Platform Migration of Software Architectural UML-RT Models: An Industrial Experience

Malvina Latifaj[1][0000−0002−2754−9568], Federico Ciccozzi[1][0000−0002−0401−1036], Antonio Cicchetti[1][0000−0003−0416−1787], and Mattias Mohlin[2]

[1] Mälardalen University, Västerås, Sweden
malvina.latifaj@mdu.se, federico.ciccozzi@mdu.se, antonio.cicchetti@mdu.se
[2] HCL Technologies, Malmö, Sweden
mattias.mohlin@hcl.com

Abstract. In this paper, we focus on the development of a model migrator tool that automates the migration of software architectural UML-RT models from HCL RTist to RTist in Code. HCL RTist is a software development environment designed for the development of complex, event-driven, and real-time applications, solely supported in the Eclipse Desktop environment. On the other hand, RTist in Code offers a broader platform compatibility, being supported in Eclipse Theia and Visual Studio Code. The model migrator ensures a seamless transition for the tool users and preserves the effort and investment of RTist users, allowing them to take advantage of the benefits of the textual Art language and web-based technologies without having to recreate models manually. Furthermore, we present a model splitter tool that enables users to partition and organize Art models after their needs. Lastly, we validate a previously proposed approach for generating synchronization infrastructures in blended modeling through an industrial use-case, and demonstrate its novel application to model migration expanding its original scope.

Keywords: UML-RT · HCL RTist · RTist in Code · Eclipse · VSCode.

1 Introduction

Software architecture (SA) is a high-level blueprint of a software system that describes its structure, components, and interactions. As such, it is an integral part of the design, development, maintenance, and evolution of complex software systems [1]. Model-Driven Engineering (MDE) is a paradigm that emphasizes models as central artifacts throughout the development process [2]. The implementation of MDE practices in SA facilitates the separation of concerns and promotes abstraction. As a result, software architects can focus on high-level design choices instead of specific implementation details.

Unified Modeling Language (UML) is a standardized modeling language [3] and the most used architecture description language in industry [4]. It provides

an array of concepts for representing various aspects of a software system, including its structure, behavior, and interactions. UML for Real-Time Systems (UML-RT) profile extends UML to capture the concepts defined in the Realtime Object Oriented Modeling (ROOM) Language and to address the needs of real-time and embedded systems [5]. To support the use of UML and UML-RT in software architecture, numerous tools and frameworks have been developed; in this paper, we focus on HCL RTist[3] and RTist in Code[4].

1.1 Problem, motivation and RTist in Code case

HCL RTist is a proprietary software development environment designed for the development of complex, event-driven, and real-time applications. Operating as an extension of the Eclipse Desktop Integrated Development Environment (IDE), it offers comprehensive support for UML and UML-RT, enabling the specification and documentation of the structure and behavior of complex application architectures. Furthermore, it uses the CppPropertySets profile to encompass all the additional properties required to translate UML-RT models into C++ applications. Despite its broad range of capabilities, RTist continues to be tightly bound to the Eclipse Desktop IDE, which is not meeting the growing demands of RTist users seeking lightweight web-based technologies.

To add support for code editors and IDEs other than Eclipse Desktop, HCL developed RTist in Code, an extension compatible with Visual Studio (VS) Code[5] and Eclipse Theia[6]. With this extension, architects and developers can develop real-time stateful C++ applications by using the textual language, Art. The first steps in creating the Art language involved defining a textual syntax designed to allow the modeling of behavior using UML-RT state machines [6]. Based on the preliminary explorations of the textual syntax for UML-RT state machines, the comprehensive Art language was later developed, encompassing both structural and behavioral aspects. As a textual language, Art facilitates readability, version control, and collaboration. In addition, it provides graphical representations of class, state, and structure diagrams.

HCL's decision to provide RTist in Code as an extension for VS Code and Eclipse Theia can be attributed to several factors. As a result of its flexibility, user-friendly interface, and a wide variety of extensions, Visual Studio Code has become a popular choice for various development tasks, including software architecture modeling. Theia, on the other hand, provides an open, flexible, and extensible tool platform that is based on modern web technologies and it allows efficient tool and IDE development. Both VS Code and Eclipse Theia share a common extension model, making extensions compatible across platforms. Additionally, both can be accessed through a web browser. Finally, they have strong user communities that contribute to their popularity and adoption.

[3] `https://www.hcltechsw.com/rtist`
[4] `https://opensource.hcltechsw.com/rtist-in-code/`
[5] `https://code.visualstudio.com`
[6] `https://theia-ide.org`

With Eclipse Theia and VS Code, advanced features such as automatic completion, text search, and code analysis can be configured to increase efficiency while coding. However, since each tool uses a different Application Programming Interface (API), implementing each feature individually poses an additional challenge. In 2015, Microsoft introduced the Language Server Protocol (LSP)[7], which standardizes communication between language servers and development tools. This protocol allows a language server, containing language-specific features, to interact with development tools through the LSP. RTist in Code utilizes the Xtext implementation of the language server API, enabling the integration of Art in VS Code and Eclipse Theia, both of which support LSP.

The main obstacle for the full adoption of RTist in Code is the migration of graphical models from RTist to Art textual models in RTist in Code. As part of this migration process, the existing graphical models stored in .emx files are converted into textual models stored in .art files. When transitioning from RTist to RTist in code, it is neither feasible nor logical to require users to manually recreate their models. Due to the time and resource commitment involved in manual migration, such a requirement would undoubtedly deter users from embracing RTist in Code. A further challenge arises when considering that the migration to Art models can span two extremes; at one end, there is the option of producing a single .art file that encompasses the entire model, while at the other end, each Art element may be allocated to a separate file, making the compilation process more efficient. The likelihood of errors increases significantly when these tasks are performed manually. Thus, enabling a seamless transition that preserves the efforts and investments of RTist users while simultaneously encouraging the adoption of RTist in Code is of paramount importance.

1.2 Paper contribution

This paper presents a *model migrator* designed to automate the migration of models from RTist to RTist in Code. This migration tool aims to enhance the user experience and to enable a seamless transition for users. Furthermore, users are provided with a *model splitter* that enables them to partition and organize Art models according to their preferences. Apart from that, the model migrator is used to *validate* a previously proposed approach for generating synchronization infrastructures in blended modeling [7], demonstrating the effectiveness of the method as well as exploring a novel application of the method for model migration, extending the method's scope.

The remainder of this paper is structured as follows. Section 2 provides background concepts and compares the approach to other related works. Section 3 presents the model migrator and the model splitter. Section 4 concludes the paper with a discussion of lessons learned and planned future works.

[7] `https://microsoft.github.io/language-server-protocol/`

2 Background and Related work

As a first step in establishing an understanding of the research landscape and emphasizing the significance of our study, this section discusses the methodology employed in this paper for implementing the model migrator, which originates within the blended modeling paradigm. Following this, it examines model migration and co-evolution approaches, focusing both on generic solutions and those that are specifically focused on UML-RT model migration.

2.1 Automated generation of synchronization infrastructure for blended modeling

Blended modeling facilitates the development of complex multi-domain systems by providing seamless multi-notation modeling [7]. The mechanisms responsible for ensuring synchronization between notations are incorporated into model transformations. Manually establishing these transformations is a time-consuming and potentially error-prone process requiring advanced knowledge of transformation languages. Moreover, any change made to the synchronized languages or notations may render the transformations obsolete. Our prior research presents an automated technique for generating synchronization transformations. This method offers users the ability to define mapping rules between two domain-specific modeling languages (DSMLs) using a textual mapping modeling language (MML). The resulting mapping model serves as input for higher-order transformations (HOTs), which automatically produce synchronization transformations [8]. While originally developed for the generation of synchronization infrastructure in blended modeling, the approach is highly versatile and can be adapted for a variety of purposes. This paper focuses on its application and validation in model migration in industrial settings.

2.2 Migration of UML-RT models

HCL RTist is an advanced version of Rational Rose Real-Time (RoseRT), a tool owned by Rational Software and released in 2000. Later in 2007, IBM and Rational Software introduced Rational Software Architect Real-Time Edition (RSARTE). RSARTE, based on the Eclipse platform, offered several advantages over RoseRT, including access to Eclipse's rich features and support for UML 2.x. [14]. In 2016, IBM and HCL introduced a rebranded RSARTE version called HCL RTist. RoseRT import wizard [14] was defined to allow the migration from RoseRT to RSARTE and HCL RTist. Models created in RSARTE could be easily used in HCL RTist without migration since both tools utilize the same model file format and generate identical C++ code. The utilization of RSARTE models is possible within the open-source Papyrus framework as well. It requires installing the Papyrus component RSA model importer in Papyrus 3.x, and it translates RSA models (.emx files) into Papyrus models, and RSA profiles (.epx files) into Papyrus profiles [11]. XMI toolkit is an add-on product for IBM Rational Rhapsody that enables the migration of models created in RSARTE into

Rational Rhapsody [12]. While these solutions have been successful in migrating UML-RT models, they are tool and language-specific and do not address the migration of UML-RT models defined in HCL RTist to RTist in Code.

2.3 Model migration and model co-evolution

In the MDE paradigm, the concepts of *model migration* and *model co-evolution* are frequently used interchangeably and are generally regarded as the result of metamodel evolution. Rose et al. [15] categorize model migration approaches into *manual specification* approaches (e.g., Java[8], ATL[9], QVTo[10]), *operator-based co-evolution approaches* [16–18], and *metamodel matching* approaches [9, 19–21]. As they point out, using general-purpose or model transformation languages to manually specify migration strategies empowers developers, but it also requires more effort as they do not capture commonly occurring patterns, such as copying unchanged elements from the source model to the target model. The same limitation applies to operator-based co-evolution. Metamodel matching, on the other hand, requires minimal user effort, but may not always provide deterministic results. Rebig et al. [22] develop a more detailed taxonomy of solution techniques for the co-evolution of metamodels and models and categorize 31 existing approaches. One might question the approach employed in this study and argue that a customized approach to model migration and co-evolution may be more effective at defining the migration strategy. The RTist and RTist in Code case, however, does not involve metamodel evolution, since the underlying languages are defined separately. RTist models conform to UML metamodel, UML-RT, and CppPropertySet profiles. These profiles extend UML concepts to make RTist suitable for modeling real-time systems. RTist in Code models, on the other hand, conform to a single compact language known as Art. While Art shares conceptual similarities with UML metamodel, UML-RT, and CppPropertySet profiles, it is a distinct and more compact language. Given the substantial differences between metamodels, implementing the aforementioned approaches may pose challenges or lead to inefficiencies. Moreover, this study also aims to validate the flexibility and versatility of our previously proposed approach by exploring its application for model migration.

3 Model migrator and model splitter

This section describes the steps involved in the implementation of the model migrator and model splitter. A prerequisite for the successful migration of models is that the languages involved should be defined as Ecore-based metamodels. The latter refer to structured data representations that are described and defined employing the Ecore meta-modeling language, an integral part of the Eclipse Modeling Framework (EMF) [23]. Fig. 1 illustrates the process.

[8] `https://www.java.com/en/`
[9] `https://www.eclipse.org/atl/`
[10] `https://wiki.eclipse.org/QVTo`

Starting from UML, UML-RT, and CppPropertySet as Ecore-based source metamodels and Art as Ecore-based target metamodel, the `uml_2_art.mapping` mapping model describes the relationship between the source and target elements, as well as the constraints driving the transformation. The defined mapping model is used as input for the HOT, generating the `uml_2_art.qvto` model-to-model transformation. Upon execution of this transformation, the graphical models originally defined in RTist are translated into textual Art models. The resulting Art models can be then imported into RTist in Code in either VS Code or Eclipse Theia, allowing users to continue their development work using their preferred environment. By this point, users who wish to continue their development process within a single .art file have reached the final step. Alternatively, users who prefer to split the generated Art model into multiple .art files can leverage the model splitter that, guided by a user-defined splitting strategy, splits the generated model into multiple .art files.

Section 3.1 delves into the details regarding the mapping model (i.e., the foundation for the model migrator) while Section 3.2 describes the model splitter. Implementation details on the MML and HOT can be found in our prior work [8].

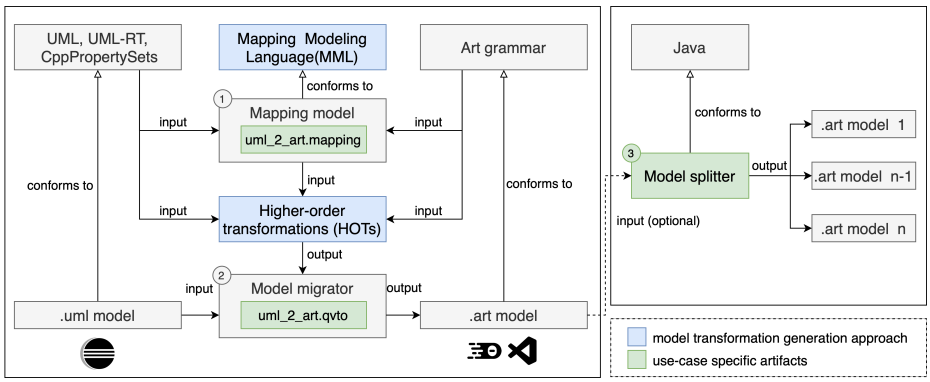

Fig. 1. Model migration and model splitting workflow

3.1 Mapping model definition

The definition of the mapping model is driven by language engineers who have an understanding of the syntax and semantics of both the source and target languages and is the sole input required to build the migration tool. Unless any of the languages involved is modified, the mapping model only needs to be defined once. The UML metamodel serves as the "base" metamodel, with its concepts being mapped to the corresponding concepts in the Art Language. UML is, however, designed to be a general-purpose modeling language applicable to various domains, making it inherently expansive and all-encompassing. Despite the significant contribution of the involved profiles to the definition of mappings, the

vastness of UML still represents a significant challenge. To tame this complexity, the mapping model is established using a systematic approach, starting with the identification of UML concepts extended by the profiles and the identification of additional UML metamodel concepts and relationships used in RTist. The set of identified UML concepts is first reviewed by language engineers and then mapped to corresponding Art language concepts. UML-RT and CppPropertySets profiles allow the extraction of information about specific UML elements that exhibit UML-RT characteristics and are required to guide the transformation. For instance, a UML class stereotyped with `umlrt::Capsule`, implies that the class represents a capsule in UML-RT. The language engineer defines conditions that identify these stereotyped classes and map them to corresponding elements in the target metamodel.

The definition of the mapping model for this particular use case required a minor customization to the MML and HOT. The main concept of MML, as discussed in [8], is the `MappingRule`. MappingRule instances, also referred to as mapping rules, establish the relationship between elements in the source and target metamodels. They can be classified into *immediate mapping rules* and *child mapping rules* depending on the type of source and target meta-elements. Child mapping rules serve various purposes depending on the values assigned to their source and target attributes. One of these purposes is to facilitate model element navigation. Specifically, when metamodels exhibit structural and/or conceptual differences, it may be necessary to traverse different paths, additional references, or intermediate elements to accurately map the source model elements to the target model elements. The implementation of MML and HOTs automates navigation to the greatest extent possible while ensuring the generation of deterministic transformations. However, attempting to cover every corner case would be impractical due to the increased complexity involved in defining the mapping model. The vastness of UML and the disparities between UML and Art necessitate intricate navigation paths combined with OCL operations for accessing references and attributes. The initial version of MML and HOTs did not support all the scenarios encountered in this case. To address unanticipated or complex corner cases without compromising the simplicity and usability of the mapping modeling language, a `VariableDeclaration` concept has been included in the MML. This concept enables users to define variables in a free-form manner (i.e., String type), capturing specific operations or scenarios that are not directly supported by the MML's constructs. Considering that this feature can serve multiple purposes, other custom transformation logic not currently supported by the MML can be incorporated as well. Upon completion of the mapping model definition, the latter is used as input to the HOTs, resulting in the generation of QVTo model transformation, execution of which leads to the generation of Art models.

3.2 Model splitter

Art models in RTist in Code can be organized according to the user's preferences. While some users choose to develop their model within a single generated .art

file, others opt for a different approach where they partition the generated Art model into multiple .art files, distributing the elements across different files.

In organizations where multiple people work simultaneously on the same model, it is common to have many files, sometimes even one model element per file (in case of complex elements). This reduces the possibility that more than one person edits the same file at the same time, resulting in merging conflicts. Moreover, with modern machines having multiple cores, model splitting is also beneficial for separate compilation. Translation of many small files concurrently is more efficient than translating fewer large files. As part of the parallelizing code generation process in RTist in Code, the C++ generator utilizes a thread pool, with each thread handling a specific .art file and producing .cpp and .h output files. In comparison, RTist parallelizes only the final stage of code generation. This indicates that code generation in RTist in Code can be significantly faster.

To benefit from the code generation capabilities of RTist in Code, a model splitter defined in Java takes the generated Art model as an input and generates multiple .art files, each containing part of the model. Splitting the model can be accomplished according to a default strategy or a custom one. In the default strategy, the partitioning process separates each `capsule`, `class`, and `protocol` into individual files. If the default strategy does not meet the users' needs, they can define a custom splitting strategy using a textual syntax. Through this syntax, they can group specific elements according to their preferences by simply defining the names of the instantiated elements (and their types if necessary).

For instance, consider the PingPong example, which consists of three capsules named `Top`, `Ping`, and `Pong`, along with a protocol named `Proto`. Specifically, the user is requesting that the Top capsule and Proto protocol be separated into separate files, while the Ping and Pong capsules remain within the same .art file. Listing 1.1 details the custom splitting strategy. Line 1 allows importing the model file and accessing the necessary data. Line 2 and 3 specify that both default and custom splitting strategies are applied to the imported model. This implies that all capsules, classes, and protocols, except those explicitly mentioned in Line 4, should be separated and stored individually in distinct files. On the other hand, the elements specified in Line 4, namely `Ping` and `Pong`, should be combined into a single file named `PingPong`.

```
1 import PingPongExample.art
2 default: yes
3 custom: yes {
4    elements: Ping, Pong filename: PingPong;
5 }
```

Listing 1.1. PingPong example splitting strategy

4 Discussion and Conclusions

This paper proposes a model migrator designed to facilitate the seamless migration of models from RTist to RTist in Code and a model splitter specifically

designed to allow users to organize their Art models into multiple files as required, providing them with greater flexibility.

The validation process for the migrator is still in its early stages. The model transformations have been tested on a limited group of RTist models with matching RTist in Code models provided by HCL. While the test suite is expanding to include more models, the ultimate validation will involve comparing the C++ code generated from migrated models with the code from the original RTist graphical models. However, this comparison is pending due to the ongoing development of the C++ generator in RTist in Code. Once it is finalized, a comprehensive validation will determine the extent to which the C++ code derived from the migrated models accurately matches the code derived from the source graphical models. Considering the separate compilation capability of RTist in Code, the latter should perform better on code generation than RTist, resulting in a faster code generation process. We are currently carrying out extensive validation of the model migrator on a larger set of models and initiating a comparative analysis between the C++ code generators in RTist and RTist in Code.

One notable advantage of the approach employed to implement the model migrator is its inherent flexibility and adaptability to language evolution. As software languages evolve and undergo revisions, traditional model migrators often struggle to accommodate these changes. Using the approach presented in this paper facilitates the adaptation of the model migrator to language enhancements by allowing the customization of mappings at a higher level of abstraction than that of model transformations. This not only saves developers considerable time and effort but also ensures the migration solution's long-term sustainability.

This study primarily focuses on the migration from RTist to RTist in Code; however, it is imperative not to confine its relevance solely to a specific user group. The proposed approach possesses the potential for seamless adaptation to diverse migration scenarios involving other pairs of Ecore-based metamodels. This adaptability can be achieved through the formulation of high-level mappings that describe the relation between the concepts of the source and target metamodels. In future work, we will explore further enhancements to the generation approach and investigate its applicability to other migration scenarios.

References

1. Garlan, D., Shaw, M. (1993). An introduction to software architecture. In Advances in software engineering and knowledge engineering (pp. 1-39).
2. Schmidt, D. C. (2006). Model-driven engineering. Computer-IEEE Computer Society-, 39(2), 25.
3. Unified Modeling Language Specification. (2017, December). Object Management Group. Retrieved June 1, 2023, from https://www.omg.org/spec/UML/2.5.1/PDF
4. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2012). What industry needs from architectural languages: A survey. IEEE Transactions on Software Engineering, 39(6), 869-891.
5. Selic, B. (1996). Real-time object-oriented modeling. IFAC Proceedings Volumes, 29(5), 1-6.

6. Latifaj, M., Ciccozzi, F., Mohlin, M., Posse, E. (2021, September). Towards Automated Support for Blended Modelling of UML-RT Embedded Software Architectures. In ECSA (Companion).
7. Ciccozzi, F., Tichy, M., Vangheluwe, H., Weyns, D. (2019, September). Blended modelling-what, why and how. In 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) (pp. 425-430). IEEE.
8. Latifaj, M., Ciccozzi, F., Mohlin, M. Higher-order transformations for the generation of synchronization infrastructures in blended modeling. Frontiers in Computer Science, 4, 2023.
9. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A. (2008, September). Automating co-evolution in model-driven engineering. In 2008 12th International IEEE enterprise distributed object computing conference (pp. 222-231). IEEE.
10. Rose, L. M., Kolovos, D. S., Paige, R. F., Polack, F. A. (2010). Model migration with epsilon flock. In Theory and Practice of Model Transformations: Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings 3 (pp. 184-198). Springer Berlin Heidelberg.
11. Papyrus Guidelines, https://opennetworking.org/wp-content/uploads/2018/08/TR-515_Papyrus_Guidelines_v1.3-1-1.pdf. Last accessed 31 May 2023.
12. International Business Machines (IBM), https://www.ibm.com/docs/en/SSB2MU_8.3.1/com.ibm.rhp.oem.pdf.doc/pdf/sodius/Rhapsody_RSART_Integration.pdf. Last accessed 31 May 2023.
13. HCL RTist, https://rtist.hcldoc.com/help/topic/com.ibm.xtools.rsarte.webdoc/pdf/RTist_RoseRT_MigrationBestPractices.pdf. Last accessed 31 May 2023.
14. HCL RTist, https://rsarte.hcldoc.com/help/index.jsp?topic=\%2Fcom.ibm.xtools.rsarte.webdoc\%2Fusers-guide\%2Foverview.html. Last accessed 31 May 2023.
15. Rose, L. M., Paige, R. F., Kolovos, D. S., Polack, F. A. An analysis of approaches to model migration. In Proc. Joint MoDSE-MCCM Workshop (2009), (pp. 6-15).
16. M. Herrmannsdörfer. COPE - a workbench for the coupled evolution of metamodels and models. Proceedings of the International Conference on Software Language Engineering (SLE) (2010), pp. 286-295
17. G. Wachsmuth. Metamodel adaptation and model co-adaptation. European Conference on Object-Oriented Programming (ECOOP) (2007), pp. 600-624
18. M. Herrmannsdörfer, G. Wachsmuth. Coupled evolution of software metamodels and models. Evolving Software Systems, Springer (2014), pp. 33-63
19. B. Meyers, M. Wimmer, A. Cicchetti, J. Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. Proceedings of Multi-Paradigm Modeling (MPM) Workshop (2010), pp. 1-13
20. A. Cicchetti, F. Ciccozzi, T. Leveque, A. Pierantonio. On the concurrent versioning of metamodels and models: challenges and possible solutions. International Workshop on Model Comparison in Practice (IWMCP) (2011), pp. 16-25.
21. K. Garcés, F. Jouault, P. Cointe, J. Bézivin. Managing model adaptation by precise detection of metamodel changes. Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA) (2009), pp. 34-49.
22. Hebig, R., Khelladi, D. E., Bendraou, R. Approaches to co-evolution of metamodels and models: A survey. IEEE Transactions on Software Engineering (2016), pp. 396-414.
23. Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). EMF: Eclipse Modeling Framework (Pearson Education).