Software Modernization Revisited: Challenges and Prospects

Hugo Bruneliere[1], Jordi Cabot[2,3], Javier Luis Cánovas Izquierdo[1,3], Leire Orue-Echevarria Arrieta[4], Oliver Strauss[5], and Manuel Wimmer[6]

[1] AtlanMod Team (Inria, Mines Nantes, LINA). Nantes. France
[2] ICREA. Barcelona. Spain
[3] UOC. Barcelona. Spain
[4] Competence Team Software Engineering (Fraunhofer IAO). Stuttgart. Germany
[5] ICT European Software Institute Division, (TECNALIA). Bilbao. Spain
[6] Business Informatics Group (Vienna University of Technology). Viena. Austria

Organizations undertake more and more software modernization projects every day, mostly due to rapid changes in the technological landscape pushing them to evolve their systems before they become obsolete. Such modernization projects are sometimes taken (too) lightly, and start more because of passing fads than motivated by real technological limitations or system problems. In our experience, many managers have a very partial view of the complexity and consequences of these projects.

A modernization project is usually composed of three phases. Firstly, reverse engineering allows understanding the system's purpose and current state. Many supporting approaches rely on model-based techniques to discover software models (UML class diagrams, state machines, workflows, etc.) representing the system at a higher abstraction level. Then, forward engineering starts where such models are analyzed and transformed (if necessary) to become the specification of the modernized system. Finally, developers and/or (semi)automated code generation techniques use these models to produce corresponding code for the targeted platform.

There are already tools that can help in some of these tasks, e.g. UML model discoverers from Java code or code generators for various platforms. But there is currently no global methodology that can guide engineers through the whole process. Morever, beyond purely technical aspects, there is no real support either for assessing project risks or final software quality.

Based on our past and present experience in conducting (and building tools for) software modernization projects, we elaborate on some important factors to consider in such projects and on a few recommendations to maximize chances of success, notably by following the ARTIST approach[7].

[7] The ARTIST FP7 project is an ambitious European collaboration involving both industrials (ATC, ATOS, Engineering, SparxSystems, Spikes) and academic/research centers (Fraunhofer, ICCS, Inria, Tecnalia, TU Wien), started on October 2012 for a 3 years duration. It proposes a complete and open technology-/vendor-neutral approach focusing on software migration to Cloud environments, including a generic methodology and accompanying base tooling support. Website: http://www.artist-project.eu

1 The Real Face of Software Migration

To get a representative vision of the factors impacting software modernization projects, we analyzed four very different case studies of real software systems (from ARTIST industrial partners) to migrate to the Cloud. These systems were *DEWS CCUI* (by ATOS), a Java system for tsunami early detection; *LoB* (by Spikes), a .NET solution for business process management; *eGov* (by Engineering), a J2EE framework to support Italian public system ; and *News Assets* (by ATC), a .NET news publication management system. After initial analysis, we came up with dimensions that helped us more systematically evaluating the complexity of each project:

- *Technical Space (TS).* A TS refers to a family of knowledge, tools and technologies used to implement the system. For instance, some software artefacts belongs to the *grammarware* TS (e.g., source code files and the grammars they conform to), others to the *xmlware* TS (e.g., XML-based configuration files and their DTD/Schemas) and so on. Each TS notably imposes a different reverse engineering approach.
- *Origin.* Some software artefacts are *generated* while others are *manually created* (e.g., scaffolding of project files derived from specification). It is important to filter out generated artifacts, for which their abstract specification is available.
- *Purpose.* We identified four main categories: *code*, *configuration*, *specification*, and *documentation*. Having a fine-grained artefact categorization is fundamental for engineers to prioritize or filter artefacts during the process.
- *Architectural viewpoint.* We observed a typical four layer classification: *presentation*, *business logic*, *data*, and *communication*. Again, this helps in problem decomposition but also to identify key company stakeholders that can assist during the migration (depending on layer(s) the artefacts belong to).
- *Environment.* External technological requirements and dependencies can strongly influence the process.
- *Size.* Size (e.g. in memory) and number of individual components are key elements to consider and study.

We circulated a survey among the use case providers and conducted interviews to evaluate each project regarding these dimensions. Table 1 shows a summary of the results at project-level, even though we realized the analysis at artefact-level. Both for the companies and ourselves, it appears to be very beneficial to get a better picture of the main project challenges (e.g., in each project a mix of TSs have to be treated and integrated properly). We would definitely recommend companies to perform a similar analysis before starting any migration project.

2 Key Success Factors

Within ARTIST, we identified several key factors which contributed at the end to a more unified, focused, goal-oriented and guided migration process. We shortly

Dimension	DEWS CCUI (by ATOS)	LoB (by Spikes)	eGov (by ENGINEERING)	News Assets (by ATC)
Technical Space	Source code (Java, Python) XML	Source code (C#, PowerShell JavaScript, HTML, CSS, ASP XAML), XML, plain graphics	Source code (Java, OWL, WSDL), XML, plain text, plain graphics	Source code (C#, JavaScript, HTML, CSS), XML
Origin	Manual code principally, some code generation	Manual code principally, some DSL-based code generation	Balanced (partial generative approach for code)	Manual code principally, few code generation
Purpose	Application, Data	Application, Configuration	Application	Application, Data, Configuration
Architectural Viewpoint	Presentation, Business Logic, Data	Presentation, Business Logic, Data	Presentation, Business Logic, Data	Presentation, Communication, Business Logic, Data
Environment	Eclipse Platform (Java), Linux OS	Microsoft Visual Studio, SQL, Server (.NET), Windows OS	Eclipse Platform (Java), Protégé (ontologies)	Microsoft Visual Studio (.NET), Oracle RDBMS
Size	Medium	Medium for GPL parts, rather small for DSL parts	Large for ontology parts, rather small for the rest	Large for the application, medium for the rest

Table 1. Summary of ARTIST case studies according to the identified dimensions.

elaborate on them here and discuss how we considered them in the particular case of the ARTIST methodology described after.

One format to rule them all: By using a model-based migration approach, where the system is represented by interrelated models (showing different views on the system), we have been able to easier deal with the heterogeneity of the different TSs. Specialized components (called "discoverers") take care of injecting the content of artefacts (from different TSs) into the common model-based migration platform (i.e., into the modelware TS). Discoverers for several kinds of grammar-based and XML-based artefacts are publicly available. For all the generated models to be interoperable, the modeling platform shares a common meta-metamodel (provided by the Eclipse Modeling Framework in our case). Once in the modeling realm, we benefited from the most appropriate mix of modeling languages: UML, SysML, BPMN, KDM (cf. the Architecture Driven Modernization (ADM) initiative[8]) as from the OMG, or more domain-specific languages (DSLs). We also relied on corresponding techniques, such as model transformation or text/code generation, to process models in a generic way.

Different views for different stakeholders: Modeling frameworks separate the visualization of modeled information (a.k.a. the concrete syntax) from its content (a.k.a. the abstract syntax). This allows processing model contents efficiently and, at the same time, providing several visualizations (possibly using alternative notations) for a same model or model fragment. This distinction is specially useful for multi-viewpoints modeling languages, such as UML or ODP, which predefine different viewpoints for different stakeholders. It makes possible to compute different views on a system to emphasize and/or neglect certain aspects. From our experience, viewpoints are beneficial within migration projects and not only for standard system design. During reverse engineering, viewpoints are very useful to improve system understanding (e.g. separating system's structure from behaviour, or architecture from detailed implementation). In forward engineering, viewpoints are also a fundamental brick to take decisions on software modifications.

Non-Functional Properties (NFPs) as first-class citizens: While migration projects traditionally focus on treating functional aspects of systems, it

[8] `http://adm.omg.org`

is more and more frequent in recent practices to deal with non-functional aspects as well. This is particularly true in the context of software migration to the Cloud. Quite often, underlying base platforms and programming languages are not changed during migration. However, new features or capabilities brought by Cloud environments can be used to improve NFPs that are important for the system and its owner. In such projects, NFPs are the main driver for migration and must be taken into account in all phases. A concrete example is the design and realization of refactorings tailored to improve performance and scalability.

Migration is a process: In addition to represent software artifacts, models are also relevant to explicate knowledge on the migration process itself. This can be exploited further to enact a well-defined migration process for a given project at hand. We have established a systematic roadmap for achieving software migration by providing a reference process (as an explicit process model) to be customized for a specific migration project. The available tooling supports this customization and allows following the specified process step-by-step. Thus, defined processes do not only concentrate on actual code migration activities. They also address pre-migration (e.g., specifying migration goals, identifying a target environment to satisfy expressed requirements) and post-migration (e.g., evaluating initial migration goals against the finally migrated system) phases.

3 A Concrete Approach for Modernization to the Cloud

We introduce hereafter (cf. Figure 1) the actual realization of the previous principles in a concrete methodology developed within the ARTIST EU project. Following the identified guidelines, the ARTIST methodology is split up into three phases:

- A pre-migration phase dealing with the evaluation of the existing software, notably in terms of migration feasibility from both a technical and business perspective (e.g., checking Cloud compliance based on standards (TOSCA) or drafts (ISO CCRA)).
- The migration phase itself, covering both reverse engineering and forward engineering activities (including possible optimizations within the migrated software). The selection of the target Cloud provider is also performed in there.
- A post-migration phase, addressing the verification and validation of the migrated software as well as the potential certification for the newly produced (Cloud-based) pieces of software, notably against Cloud standards and current best practices.

We have implemented two versions of the tooling supporting this methodology, one relying on the Eclipse framework and another based on SparxSystem Enterprise Architect (to deal respectively with both the Java/JEE and C#/.NET cases). All along the process, but more particularly during the migration phase and optimization actions, we are extensively using UML models (sometimes extended with some profiles) to represent the different required aspects of the system. This also facilitates model exchange between different tools and the building

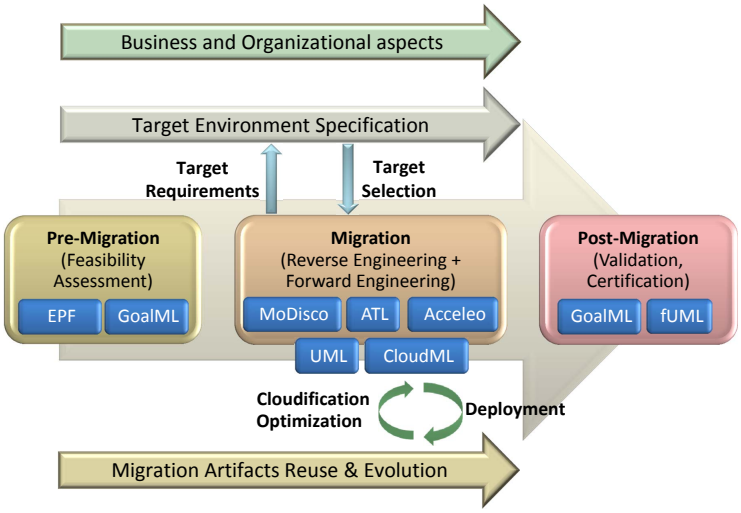

Fig. 1. ARTIST Migration Process

of several viewpoints. Complementarily, all used and/or produced models can be stored and retrieved from the so-called ARTIST repository at any time during the process.

Other modeling languages and techniques are employed in more domain-specific tasks. To mention a few, we are using a Goal Modeling Language (GoalML) as part of the pre-migration phase in order to precise the scope of the intended migration (e.g., as far as non-functional properties are concerned) and help in determining whether it is actually feasible or not. In the post-migration phase, fUML behavioral models of the deployed system are automatically checked against the initial goal models to validate if corresponding objectives are satisfied. The target specification and deployment process itself depends on the CloudML language (expressed as a UML profile) to represent required cloud infrastructures.

All this is made easier by the reuse of open source modeling solutions (thanks to the shared modelware TS), mainly coming from the Eclipse ecosystem. We can notably mention the Eclipse Process Framework (EPF) (in charge of enacting and customizing the ARTIST Methodology Process Tool to follow the various ARTIST phases), the MoDisco model driven reverse engineering framework (for code-to-model discovery), ATL (for model-to-model transformation) or Acceleo (for model-to-code generation).