FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Investigation of Data Modelling Approaches for the Integration of Heterogeneous Engineering Tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Florian Waltersdorfer
Matrikelnummer 0526618

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dr. Stefan Biffl
Mitwirkung: DI Dietmar Winkler

Wien, 21. Januar 2014

(Unterschrift Florian Waltersdorfer)

(Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Florian Waltersdorfer
Kupkagasse 6/18, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Florian Waltersdorfer)

Abstract

The development of large automation systems, like power plants or steel-works, requires the combined efforts of engineers from several disciplines, such as software, electrical, and mechanical engineering. Engineering groups working together have common concepts that are, however, represented in different ways in the tools used by each group. These tools are hardly interoperable, and aimed at distinct tasks (for each engineering discipline), so their data models are often highly heterogeneous, causing "semantic gaps" between the groups of engineers. This term refers to the fact that exchanging data between different groups needs to be done manually, as some parts of the common data model are simply missing in some tools. These missing parts have to be added either via custom fields in the specific tool (i.e. fields with no particular meaning to that tool) or after the data exchange by editing export/import files "outside" of the tools.

Designing an integration solution to ease these development processes requires application integration developers to start by re-modeling the common data models mentioned above in a way that allows an efficient data integration to take place. Said developers and application integration power users are the main stake-holders, as they usually do not have modeling guidelines for their integration solution and neither designs, nor an efficient way to evaluate a given solution.

To allow for an efficient and effective integration solution, a design guideline is needed that results in similar (possibly equal) data model designs even if executed by different application integration developers. In addition, a method that allows for identification and correction of common errors and known pitfalls before implementation after the design is desired.

By iterative prototyping, candidates for modeling styles are tested for feasibility, while inspection-based techniques and the Architecture Tradeoff Analysis Method (ATAM) will be used to adjust a feasible solution candidate and to determine a design guideline to aid the initial creation and the change requests against such a model during its life-cycle.

Criteria for a good solution include:

- ease-of-use method to design and validate data integration models
- robustness against later updates of the data model
- high rate of detection of common design errors and invalid specifications

Kurzfassung

Die Entwicklung von großen Automatisierungssystemen, wie z.B. Kraftwerke oder Stahlwerke erfordert gemeinsame Anstrengungen von Ingenieuren aus verschiedenen Disziplinen wie Software Entwicklung, Elektrotechnik, und Maschinenbau. Ingenieursgruppen die zusammen arbeiten, haben gemeinsame Konzepte, die in in den Werkzeugen jeder Gruppe vertreten sind, aber verschieden dargestellt werden. Allerdings sind diese Tools nur selten interoperabel, die diese auf bestimmte Aufgaben (für jede technische Disziplin) ausgerichtet und ihre Datenmodelle oft sehr heterogen sind, was zu "semantischen gaps" zwischen den Gruppen der Ingenieure führt. Dieser Begriff bezieht sich auf die Umstand, das der Datenaustausch zwischen verschiedenen Gruppen manuell durchgeführt werden muss, da Teile des gemeinsamen Datenmodells in einigen Werkzeugen fehlt. Diese fehlenden Teile müssen entweder über benutzerdefinierte Felder in das spezielle Werkzeug (d.h. Felder ohne besondere Bedeutung dieses Werkzeug) hinzugefügt werden oder "von außen", nach dem Datenaustausch durch Editieren Export/Import-Dateien.

Der Entwurf einer Integrationslösung, um diese Entwicklungsprozesse zu erleichtern erfordert es, dass Integrationsanwendungs-Entwickler die zuvor erwähnten gemeinsamen Datenmodelle so neu designen müssen, dass eine effiziente Datenintegration stattfinden kann. Diese Entwickler und Power-User der Integrationsanwendungen sind hier auch die wichtigsten Akteure, da sie in der Regel weder über Modellierungsrichtlinien für ihre Integrationslösungen verfügen, noch über eine effiziente Möglichkeit, eine bestimmtes Design zu bewerten.

Um eine effiziente und effektive Integrationslösung zu ermöglichen, ist daher eine Designrichtlinie notwendig, die zu ähnlichen (idealerweise identen) Datenmodellen führt, wenn sie von verschiedenen Anwendungsintegration-Entwicklern angewendet wird. Ebenso ist eine Möglichkeit erwünscht, die das Erkennen und Korrigieren häufiger Fehler und Pitfalls vor der Umsetzung erlaubt.

Durch iteratives Prototyping werden mehrere Modellierungsstile auf ihre Machbarkeit getestet, während Inspektions-basierte Techniken und die Architecture Tradeoff Analysise Methode (ATAM) benutzt werden, um die praktikabelste Lösung anzupassen und eine Design-Richtlinie abzuleiten, die das anfängliche Design unterstützt und change requests gegen ein solches Modell während seines Lebenszyklus ermöglicht.

Kriterien für eine gute Lösung sind:

- eine Guideline zum Erstellen und Validieren von Integrationsdatamodellen
- Robustheit gegen spätere Anpassungen des Datenmodells
- hohe Erkennungsrate bekannter Design-Fehlern und ungültigen Spezifikationen

Contents

CHAPTER 1

Introduction

This chapter explains the motivation behind this work, its concrete challenges, goal, used methods and its inner structure. First, the initial scenario leading to an ineffective integration approach is described to understand the reasoning behind this work and its spadework. Second, the introduction details the challenges of finding an efficient and effective solution and the constraints which have to be taken into consideration, to make it clear why previous solutions have to be extended and/or modified. Third, having the research issues specified, the expected outcome to tackle them are listed (within the limits of this work). Fourth, the way of finding a solution and validating it against the requirements defined beforehand is explained to illustrate how the proposed solution was found. And last, this chapter also contains an overview of the remainder of this work, providing a short outlook for each chapter.

1.1 Motivation

The development of today's automation system, like power plants or steel-works requires the combined efforts of engineers from several disciplines such as software, electrical, and mechanical engineering. These groups have different perspectives on their shared work: Mechanical engineers are tasked with designing and planning the layout of the larger machinery (e.g. turbines, pipes and pumps) and entire building blocks (e.g. the dam holding the water or reserve tanks), having to worry about if their construction withstands the intense physical stress it takes during operation. Electrical engineers design the wiring and placement control hardware of these large-scale components as well as the transportation of energy (as a large factory either produces or consumes high amounts of energy), having to deal with safety regulations (e.g. emergency stops) and efficiency constraints. Software engineers have to program or customize the software placed in the control hardware, implementing the desired behavior of this large-scale system via a series of small, inter-locked components. Since software engineers are the last in this chain of work, they receive a relatively precise set of requirements and constraints, so their main concern is to satisfy the initial (software) requirements within in the limitations of their colleagues' spadework.

Although working separately most of the time, engineering groups have common concepts that are represented in many ways in the tools used by each group. However, these tools are rarely interoperable and aimed at distinct tasks (for each engineering discipline), their data models are often highly heterogeneous, causing so called "semantic gaps" between the groups of engineers. This term refers to the situation that exchanging data between different engineering groups needs to be done manually, as some part of the common data model is simply missing in some tools. These missing parts have to be added either via custom fields in the specific tool (i.e. fields with no special meaning to that tool) or after the data exchange by editing export/import files "outside" of the tools.

A common scenario illustrating the issues with these gaps is handling change requests when engineering industrial power plants. During the engineering process, the planned level sensor of a pump is replaced by a sensor of different type, due to lower cost or better performance. The engineer performing or supervising the change adapts the electrical plan to document the replacement. This change however, has potential side effects on the long term, since on a higher level, the control software of the pump now receives different values, e.g. with its values inverted. In turn, the engineer responsible for the control software now would have to update the code by reversing boundary checks. Due to security regulations, the software engineer has to be informed of the repair anyways, but has no immediate possibility of determining the impact of this change, since no information about the relation between sensors and the software is available.

This re-occurring process during and after the development of larger systems follows a structured model, e.g. the V-Model XT [12] [8] or VDI guidelines [75] in most companies (even if not "officially" present, same kind of standardized process will be established over time). However, due to the semantic gaps, the units of work (e.g., written specification, circuit diagrams, code fragments or simulation configurations) have to be "translated" into the data model of the receiving group. In addition to the fact that these transfers are costly and error-prone (as they are keeping the engineers involved from their actual work), they are supposed to be one-way and happen only a few times during a project, which in terms of iterative, perhaps agile, development is simply not the case.

Therefore, an integration solution to allow for continuous data exchange, storage and versioning would pose a viable way of dealing with the issues arising from semantic gaps.

1.2 Problem Statement

By iterative prototyping of solution candidates (for data integration and tool chain support) by the CDL-Flex for affiliated research partners, different modeling styles were tested over the course of several months. During each iteration, a small set of use cases (2-4) regarding tool integration was taken for testing the current development iteration. Using the results from previous iterations, more detailed specifications of those use cases were created, combined with sample data to "simulate" runs of said use cases. Using this new information, the application integration development team at the CDL-Flex re-designed current tool data models and modified or extended the business logic. The newly deployed integration solution is bundled together with a set of automated GUI tests [73], so that users can validate that the specified use cases

were implemented and how they can execute them. Depending on how different the use cases (or parts of them) were in comparison to previously solved challenges, changes to the underlying architecture were made. These architectural changes also affected the way in which tool data and common data were stored, transformed and treated "inside" the integration solution (e.g. if the actual model instance of common tool data is treated as a first-class-citizen or mere configuration), introducing a set of different data modeling styles for ostensibly similar scenarios. Some of these styles (e.g. ESB-like [13] and VCDM [77]) proved to be quite feasible long-term solutions for the immediate purpose of model integration and tool chain support. However, through regular user feedback and iterative evaluation of the project's requirements over time, members of the laboratory discovered that changes to the common data model happened more frequently then expected. These changes often caused either the redeployment of a large number of components, intense re-configurations or flat-out errors in the solutions due to their low tolerance against changes to the common data model. This resulted in frequent, heavy changes to a server system which, by that time, is regarded as a critical asset for the success of an engineering project. As these changes caused down-times and required additional effort to both implement them and train users to understand them, the loss of productivity caused in this way was a major risk to the success of the acceptance of our integration solution. In addition, introducing such changes requires the development team to undergo a tedious process to design, implement, integrate and (after initial user testing) fine-tune it.

Such changes come from a series of reason: Invalid/incomplete design at the beginning, tool/process changes, variations between companies/departments and the demand for new features of the integration solution not considered before, such as claim management. From these testing results, three main concerns were identified.

1. The need for a flexible and inter-changeable data model description format, which can be integrated into a system for automated data exchange between the tools, but is not directly integrated into the main business process (for easier, side-effect-free modification).

2. Once such a solution is deployed, it should remain stable in terms of user experience and not require major updates in short intervals.

3. As these changes are often quite trivial (such as adding one new field or increasing the length of it), power users of such an exchange system should be able to design such model updates themselves with little to no programming skills.

1.3 Aim of the Work

The main outcome of this work will be to find or create a modeling approach for data integration of third party data models in heterogeneous engineering environments. This data model design guideline should support integration application developers on their task to integrate one or more third party tools into an existing infrastructure or to plan a new one from scratch. Answering a set of questions will allow integration application developers to validate the design and/or extended tool domain data models and common data models for cross-discipline work-flows as

well as their respective conversions into each other (if needed). The main motivation behind this guideline is to

- determine whether the approach is suitable for a specific scenario,
- ease the initial modeling effort by avoiding known mistakes and ”smells“ from previous integration projects
- allow for later modifications of the data models without breaking the initial design.

The second goal of this work deals with the validation of that approach itself. Since it is a combination of existing styles and new concepts from the CDL-Flex, its effectiveness is not an established fact. So, this design guideline's effectiveness is evaluated by applying it to design and implement a specific use case involving three engineering disciplines with their respective tools (using the Open Engineering Service Bus). Regarding the efficiency of the guideline, a comparison with other approaches, being either known best practices or alternative designs, will be shown to illustrate its strength and weaknesses in the given scenario(s).

In short, major results are:

- a data modeling approach for application integration in heterogeneous engineering environments
- an inspection framework / guideline for validating deployed data models and later changes
- an evaluation of the modeling approach using the Open Engineering Service Bus framework

1.4 Structure of the Work

The remainder of this work is structured as follows:

Chapter 2 consists of two major blocks: Known solutions and required tools. It describes state-of-the-art solutions for integration scenarios and current best practices concerning data integration. It also introduces the concepts of “Enterprise Application Integration” (EAI), related state-of-the-art methods and spadework from the CDL-Flex. In the second part, the set of tools and concepts needed to design and evaluate a solution against other architectures and constraints from the given use case are explained.

Chapter 3 extends the initial goals to this works to the “full” research issues, using the insights gained from the initial literature research. In addition, this chapter highlights the specific challenges for each goal and why they need to be dealt with by a good solution.

Chapter 4 details on the methodology to come to the suggested solution in chapters 5 to 8. Using the findings from chapter 2 the procedure to find solutions to the questions raised in chapter 3 is given.

Chapter 5 illustrates the prototyping and user testing phase for an initial candidate solution and its refinement to produce the final solution candidate.

Chapter 6 deals with the evaluation of the solution candidate for a data model design style from chapter 5. The chapter details the deriving of the guideline for validating and assisting the data modeling.

Chapter 7 handles the drafting of a validation processes for a given data model design instance. Also, the extension of this set of questions to handle updates of implemented data models during the life-cycle of an integration solution is demonstrated in the same manner.

Chapter 8 uses the results from chapter 6 to illustrate a possible data integration style and how to derive the actual data model from the initial specification. In addition, the procedure build in chapter 7 and its extension are provided in full detail there.

Chapter 9 summarizes the results to show that the proposed solution is an effective and efficient solution to the initial challenges that motivated this work. The second part of this chapter highlights unsolved challenges or newly discovered ones, showing the limitations and scope of the solution.

Chapter 10 concludes the findings from the previous chapters to allow for a fast insight into this work's outcome. In addition, candidates for future work are given, based on unsolved challenges and the use of the solution in the future.

CHAPTER 2

Related Work

This chapter consists of two major blocks: Known solutions and required tools. In the first part, a description of state-of-the-art solutions for integration scenarios and current best practices is given. This includes a short description of the concepts of "Enterprise Application Integration" (EAI), its widely accepted solution approaches and previous work from the CDL-Flex, based on formerly mentioned best practices. In the second part, there is an overview on the set of tools and concepts needed to design a solution and to evaluate it against other solutions and our initial requirements.

2.1 Enterprise Application Integration

Even though this work focuses on the data integration part and guidelines for data models, an overview of application integration and its different levels is useful to explain some of the currently used approaches for data integration and to give a little insight in today's tool landscape (in the automation industry). In "Enterprise Application Integration" [48], four stages of application integration (which are not mutually exclusive) are listed:

1. Data Level
2. Application Interface Level
3. Method Level
4. User Interface Level

While these stages alone do not allow one to rate the maturity of an integration solution, they provide for a good estimate on how sophisticated such a solution can be.

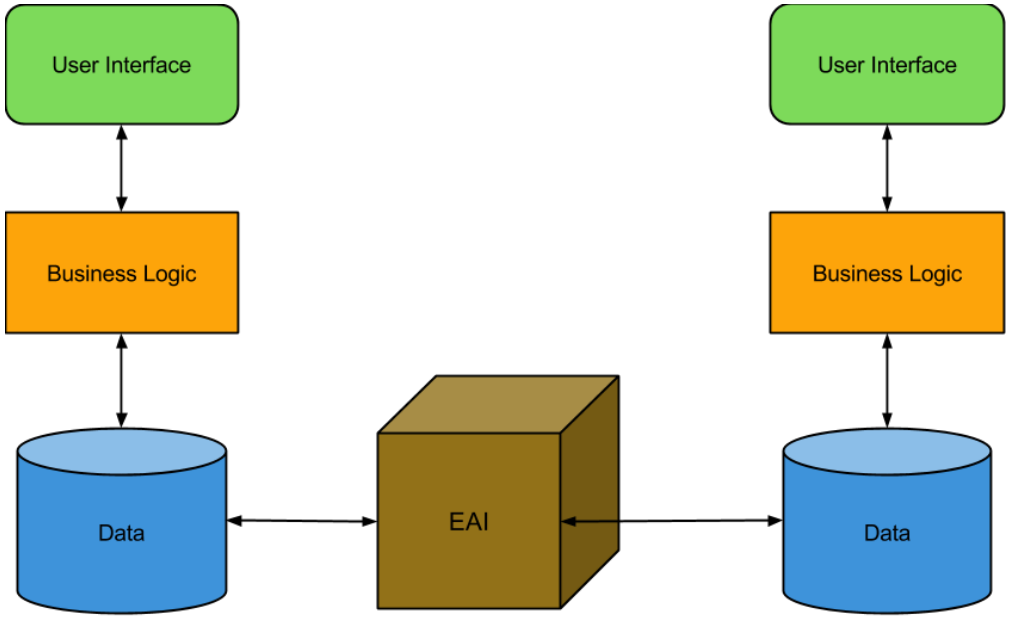


Figure 2.1: Data Level Integration [48]

2.1.1 Data Level

Integration on a data level refers to a scenario in which the "integrated" applications are not addressed directly by the integration solution, but via their data sources, see Figure 2.1.

Any interaction between the tools is mapped to specific data exchanges between their databases (like automated data im- and exports), while business processes and new "features" are realized by third party tools which feed new data to the tools' databases or even manipulate their schemata. Using this approach usually requires a very detailed understanding of a tool's database schemas, how the database is used during runtime and when/how it is safe to access. However, there is a series of reason why this approach may be applied. As stated in "Data Integration Blueprint and Modeling" [29], data level integration is considered as the most basic form of application integration and can serve as a starting point for future integration efforts [82] [31]. One reason being that the software to integrate does not provide any other means of access (monolithic software without any interfaces), which is quite common for older software. Another, less technical reason is that the company/developer maintaining the software is not cooperating with a desired integration project for several reasons:

- Pricing issues, i.e. that API access or programmatic extension of the software is considered too expensive (for the customer)
- Threatening the business model, i.e. that the company providing the software does not wish features to be accessed from outside their tool and/or automated

- The company does not exist anymore or does not have the know-how to modify the software in the desired way
- The company does not know of the integration efforts and is deliberately kept in the dark, e.g. in case of another software vendor promoting integration with that software without an explicit cooperation
- Data integration is the "only" form of exchange companies, or entire industries can (yet) agree upon

A noticeable variant of this integration style is the linking of copies of the same software in this way, e.g. *EPLAN Electronic P8*[1] and *SICAM TOOLBOX II*[2]. In this cases, several engineers share their work by accessing the same database to store their work, although the initial design of these applications did not enforce this behavior (one database per copy of the tool is also considered a viable configuration). Whether as a consequence of user demand for open systems or by examples from other industries, such as telecommunication [30], the automation industry tends to move more towards standard open data formats [5] [1] over the course of the last few years [21] [20].

2.1.2 Application Interface Level

Application Interface Level Integration refers to using the tools' provided external access (API) to link applications together and extend their capabilities, see figure 2.2, as current software development best practices highly embrace open and reusable software architectures [27].

In addition, the idea of opening the API of one's software to other participants has become a (more) popular concept in different fields, like telecommunication [61] perhaps emerging from the previous "Web 2.0". trend [69] In contrast to Data Level Integration, this method usually happens in consent with the tool's maintainer (as they exposed the interface) and allows for a more stable tool integration, yet is limited by the same interface.

Currently, the perhaps most common use of API level integration in the automation industry is in tool suites: Developed by different departments and sometimes former separate companies, e.g. the *SICAM TOOLBOX II*[3], tools suites combine software for different tasks that share a common work-flow and/or require access to shared base data, e.g. stock inventory, program libraries or document storage.

[1] `http://www.eplan.de/de/loesungen/elektrotechnik/eplan-electric-p8/`

[2] `http://www.energy.siemens.com/hq/en/automation/power-transmission-distribution/sicam-substation-automation/sicam-1703/sicam-toolbox.htm`

[3] `http://www.energy.siemens.com/hq/en/automation/power-transmission-distribution/sicam-substation-automation/sicam-1703/sicam-toolbox.htm`

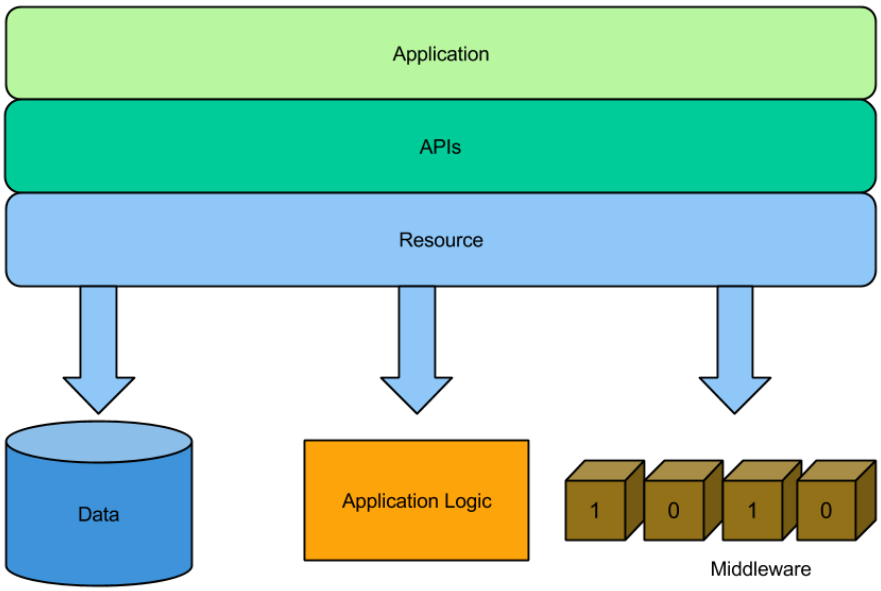


Figure 2.2: Application Interface Level Integration [48]

2.1.3 Method Level

Method Level Integration refers to a much "deeper" and more invasive coupling of application then API Level Integration, in the way that the same methods and/or functionalities a software uses internally are exposed to outside tools and that said software is actively adapted to the needs of an integration solution. Requiring a very close cooperation of the partners building an integration solution and most certainly heavy refactorings of the affected software, this degree of integration may not be efficient in all cases [48]. This approach is strongly motivated around the idea of software reuse [52] [54] and composition [6]. Following the DRY-principle [70] on a higher level, Method Level Integration aims to exploit a tool's existing features to couple it with one or more other tools. Some research effort has been made to support [74] [41] and measure [28] the development of extendable software with "clean" architectures [67], making integration on this level a more feasible option, as compared to earlier stages of software and data integration [32].

Still, with the higher risks and costs of this method compared to the other three methods, Method Level Integration is likely to remain a supplement, but will rarely be the main integration approach when planning a solution for heterogeneous engineering tools (from multiple vendors).

2.1.4 User Interface Level

User Interface Level Integration [58] itself is a style very different from the other three and yet more restricted in most cases. The big difference of this method lies in the fact that the goal of this integration method is not to somehow connect parts of the software into a larger system, but to simulate user interaction or forward user interaction to the software. Neither the

behavior of the "connected" software is not expected to change, nor the way it is used, so it is still treated as an isolated system to a much higher degree then in any of the other methods. Following this line of thought, restricted refers to the fact that using only the user interface of a tool limits the possibilities of adding new capabilities. In comparison, even with "only" Data Level Integration, an application can be "tricked" into a behavior not initially intended, such as sorting and filtering data, manipulating another tool's data or reacting on changes (via database triggers). On the other hand, the ways of achieving User level Integration can be done in many more ways then the other integration types, as the program which receives the (simulated) user interaction does not influence the integration solution's mechanics.

Although User Interface Level Integration is less flexible and most certainly requires more effort, sometimes it is the only way to access a tool or at least some of its features, and has seen a recent boost in popularity with the past upcoming of web mash-ups [34] [3] [81] and efforts to reuse this trend for EAI [49].

Common uses of this integration approach are:

- Web Mash-ups which aggregate components of multiple websites into one portal/mash-up solution
- Tool Suites which execute a series of tools from a unified UI placed on top of the other tools' UI, sometimes allowing for a shared clipboard or sending specific hotkeys/macros from one tool to another
- Remote Desktops allowing a tool to be used from a workstation not supporting the specific tool, e.g. due to its operating system, licensing or performance issues
- Terminal Emulators providing access to exotic and/or legacy mainframe systems that run only that specific tool

2.2 Data Modeling and Integration

Since the importance of data integration [46] and its challenges [9] have been elaborated, much effort has been spent to solve data integration challenges in a variety of fields, such as E-commerce [50] [15], Healthcare [11] and Bioinformatics [47]. Despite their usefulness, such data integration efforts are treated independently of other underlying application integration issues with the systems they are deployed onto. As indicated before, designing and implementing such an integration solution, especially in an enterprise environment, is a complex process with little to no standard solutions available. Yet there exists a series of guidelines, common terminology and well-defined patterns to deal with most known scenarios on an abstract level [38] [36] as well as with concrete technological environments [53] [39]. Concerning data integration "inside" such solutions however, the degree of abstraction of data integration leaves much room for details about how to connect and/or combine heterogeneous data sources. Outside the scope of EAI, works dealing with data integration separately, also retain a relatively high level of abstraction, either focusing on the granularity and frequency of the data integration and its use [63] [66] and on the technical means to access and combine data sources [2].

2.2.1 Data Integration Styles

In data integration, the (perhaps) best classification to be found are the following (maturity) stages, taken from "Data Integration Blueprint and Modeling: Techniques for a Scalable and Sustainable Architecture" [29].

- Federation (of Databases)
- Extract, Transform & Load
- Enterprise Application Integration
- Service Oriented Architecture

This classification also is the most concrete, technoloy-independent listing of data integration strategies. The lack of more concrete guidelines and/or patterns can lead to vastly different solutions for (relatively) similar problems. This poses a high uncertainty for planing application integration projects, since the effort for designing the data model and its mappings to local data sources cannot be estimated and hardly validated, in comparison to other parts of the integration solution, such as messaging and application adapters/connectors. Below follows a short introduction to these (ordered) stages of data integration, illustrating the connection between application and data integration.

Federation Historically the first of the four stages (Chapter 1. "Types of Data Integration" [29]), Database Federation is a pure Data Level Integration approach with a "federating" software providing uniform access over multiple disparate and possibly distributed sources, e.g. as in figure 2.3.

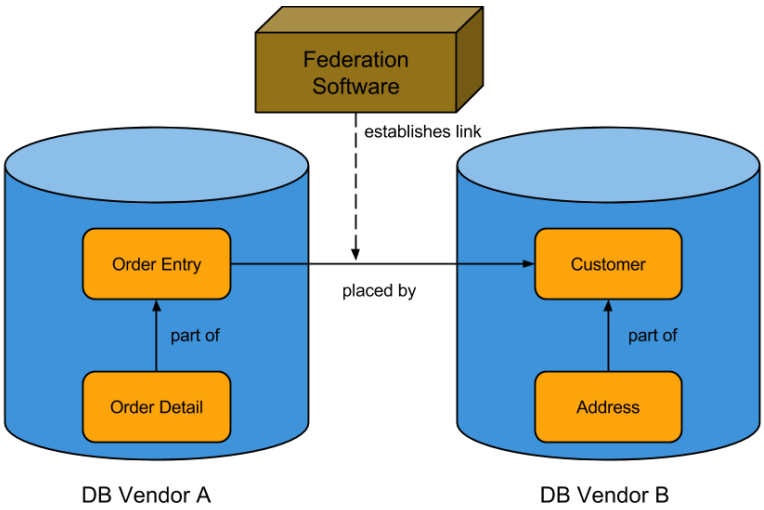


Figure 2.3: Federation [29]

The notable distinction compared to the “Data Level Integration” approach lies in the specific way its outcome is used. The federation software acts like a new, independent tool alongside other tools and data sources and (should) behave like any other application and/or database in terms of performance and accessibility. While this solution usually adds new features to an environment, it also adds a new (virtually) isolated tool, which may be unadvisable in an already scattered environment with heterogeneous (engineering) tools.

Extract Transform & Load The next stage of data integration, ETL, details a system in which data from various sources is actively collected, modified and copied in another data storage for later analysis, monitoring and/or reporting, as in figure 2.4

The more common term for this method is “Data Warehousing” [14], which has become a necessity for larger companies [79], regardless of their supplementing data integration plans. Unlike the other three methods for data integration, the main goal of ETL is not to extend capabilities of tools used in the day-to-day business, enhance “real-time” interactions or change user’s work-flows (directly). One of the most complex challenges of DW does not deal with the collection of vast amounts of data, but with the correct filtering and representation of it, enabling experts (in both technical and economical questions) to receive insights from the “big picture”, that would go unnoticed in the day-to-day business. As stated above, Data Warehousing is used to aggregate data from other tools and allows for a more “high-level” view of the “operative data”, e.g. for quality assurance [16]. For the same reason, in an application integration project, Data Warehousing can be a requirement and key feature, but does not affect the architectural style of the integration solution [68].

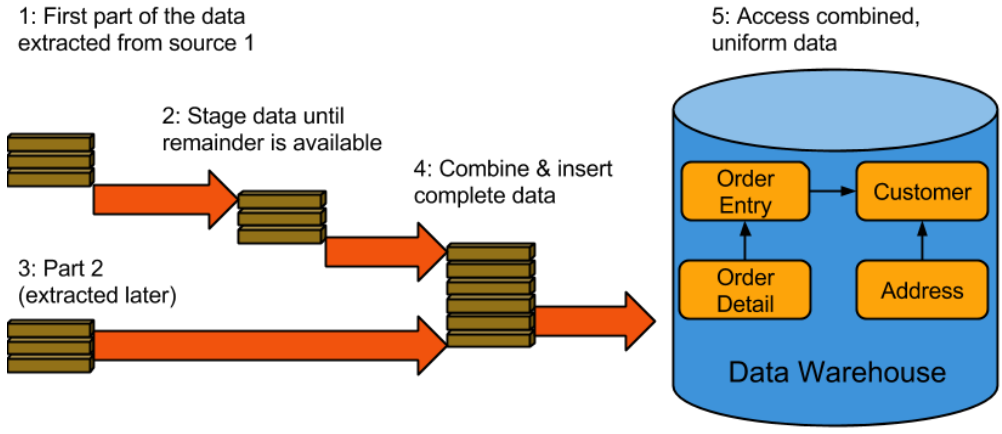


Figure 2.4: ETL [29]

Enterprise Application Integration Quite surprisingly, EAI itself is treated as a specific technique or stage of data integration in sources [2, 29, 63], which illustrates the overall issue (of a lack of concrete data integration strategies) we are facing (see figure 2.5 below).

From the application integration point of view, data integration is a sub-goal to be achieved be unspecified methods, while from the data integration perspective, application integration either dictates how data integration is done or should bring in roughly specified "tools" that provide for the required data (such as federation software or ETL engines). In most cases, EAI from the data integration perspective is considered to happen either on a method or API level: Applications across a large company are somehow connected with each other, exchanging data in suitable formats for every specific use case, but are otherwise closed systems from the outside.

Service Oriented Architecture Service Oriented Architecture depicts an integration state in which each participating tool and data source exposes its functionalities and data in globally defined, strict formats, so that a central orchestration instance can execute and control the business processes [18] [35] (figure 2.6 below).

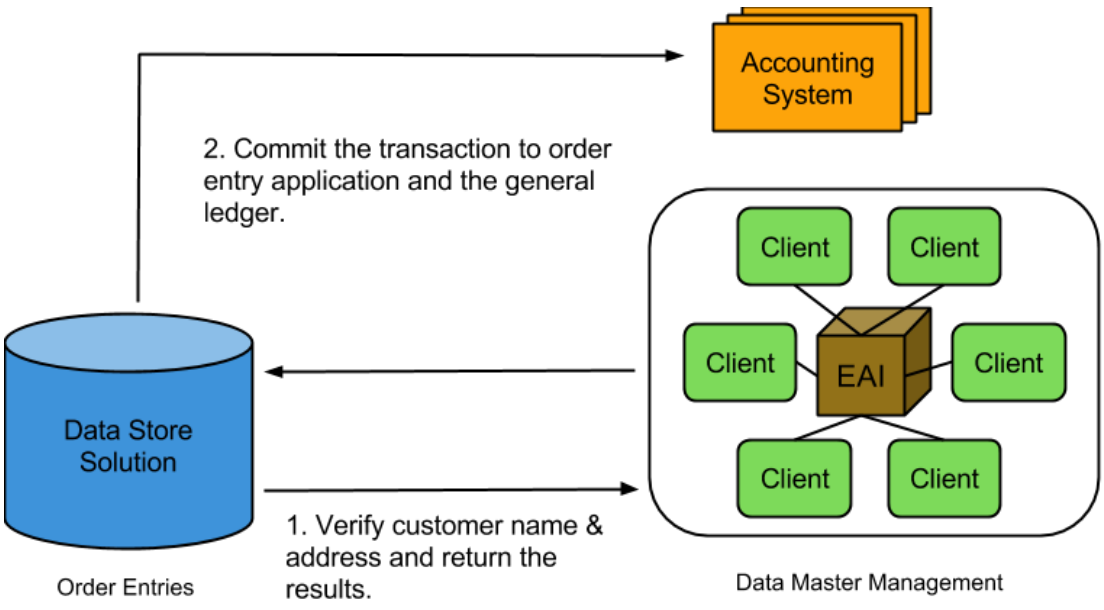


Figure 2.5: Enterprise Application Integration example [29]

In vocabulary of application integration, SOA translates to more restrictive version of API Level Integration [65], as it relies on precise standards for data mappings, service discovery and control [78]. However, even in its inherent restrictiveness, SOA does not regulate data integration in a very detailed way, nor does it provide a data integration strategy by itself. SOA only requires that data formats are compatible between the endpoints orchestrated for interaction and many frameworks, like JAX-WS[4], IBM WebSpere[5] or ASP.Net Web Services[6], provide transformation engines or allow for an easy inclusion of such. Thus, in terms of data integration, SOA (as EAI) does not narrow the modeling style for an integration solution or favors a specific style.

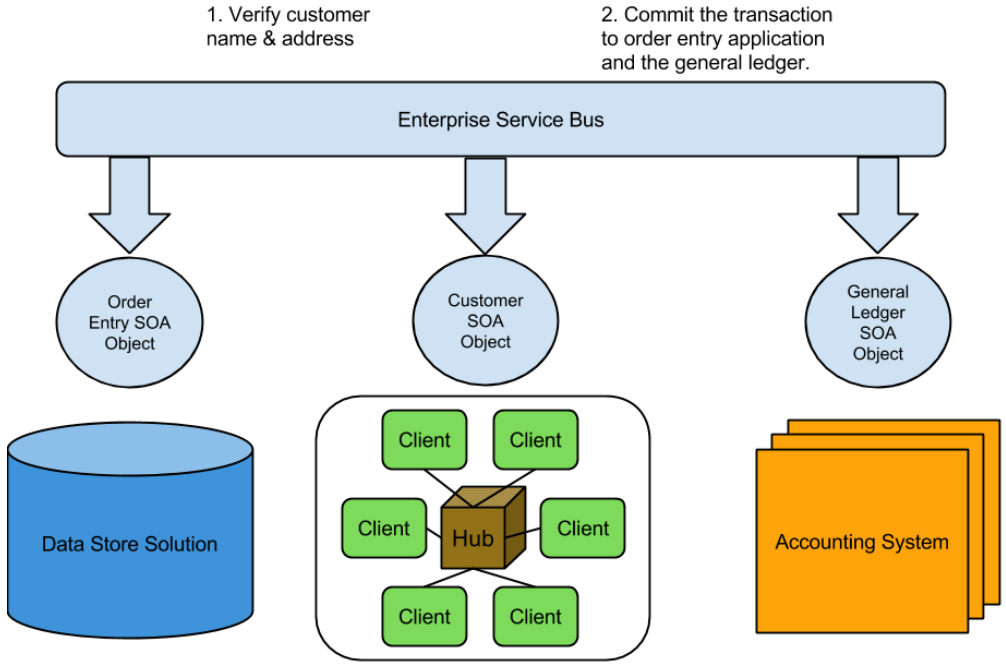


Figure 2.6: Service Oriented Architecture example [29]

[4] http://jax-ws.java.net/
[5] http://www-01.ibm.com/software/websphere/
[6] http://msdn.microsoft.com/en-us/library/aa979690(v=office.14).aspx

2.2.2 Data Modeling Styles

As explained in the (sub)sections above, unlike as for the coding part of software engineering, there is no set of patterns to model a data integration solution, however there is a set of (basic) styles about the basic structure of an integration model. There are three commonly known and accepted styles, with the first two defined in the early stages of data integration [46] [24] and a third one which was added as an extension/advancement later on [51] [55]. These styles are known as:

- Local as View
- Global as View
- Global-Local as View

Local as View The basic assumption behind LAV is that all data sources with their local schemas are more prone to change then the global schema against which they have to be integrated. Source models are treated as variants of a global model, allowing for an easy extension of the system, as long as the global schema remains "intact". In this case, a new local source means simply adding another view onto the global schema. The downsides of this style comes to light if either the global schema is unstable (i.e. due to changing integration requirements) or if the local sources are the true drivers for a data model, which is likely to be the case when using software that cannot be modified on this level.

Global as View The basic assumption behind GAV is that all data sources with their local schemas are more stable then the global schema against which they have to be integrated. All parts of the global model are treated as variants of a local models, allowing for an easy aggregation of the data from many local sources, as they are already configured in the mappings. Adding a new type of local source model to a GAV-modeled integration solution however, can result in a much higher effort.

Global-Local as View A weakness of both modeling styles comes from the scenario that the local data models differ in the level of detail, i.e. to what granularity data is stored and mapped in each local model. Such variation in granularity can either be "flattened" on the global level with GAV or on the local level with LAV, but not in both places, affecting the architecture of the software developed as the application integration solution. When integrating engineering tools for different disciplines which by definition deal with varying perspectives on a project, this can be a considerable issue. Another possible challenge with using GAV or LAV is the fact that even if all local and global models are "clean" and normalized schemas, they still can model the same or similar concepts in greatly different ways, e.g. by using different hierarchies to search, order and sort their units of work. Thanks due to basic standardization attempts for addressing elements when building automation facilities (KKS[7][8], this is a minor concern for the scope of this work.

[7] http://www.kronebach.com/kks/index.html
[8] http://www.vgb.org/db_kks.html

Nonetheless, with both GAV and LAV not always being a feasible approach, a hybrid style was defined: Global-Local as View. In GLAV, both the local and the global model are treated as views (regardless if one or both are stored as actual data base instances) in the way that each element which is query-able from one of the views can be queried from the other view as well. This is achieved by using the model for GLAV itself to express the mappings between the local sources and the global mediation scheme. The main downside of this approach lies in its much higher complexity to model, implement and maintain a solution.

2.2.3 Applications of Modeling Styles

Based on the design paradigms mentioned above, two integration application solution designs and their respective data modeling styles recently tested in the CDL-Flex are highlighted, the Virtual Common Data Model (VCDM) and an Enterprise Service Bus (ESB) based approach.

A Recent Solution Attempt: VCDM Based on a LAV modeling style, a variant to integrate heterogeneous data sources and derive new functionality from these sources, i.e. for versioning and quality assurance has been developed in the "Christian Doppler Laboratory Software Engineering Integration for Flexible Automation Systems", named "Virtual Common Data Model" [76] [72], see figure 5.1. Over the course of two years, this approach has proven itself successful in selected integration projects, however due to its prototypic nature, it lacks a formal description on how it can be applied on a new scenario. Also, since VCDM was initially designed as a solution for one very specific scenario, it lacks a way of determining if it is a fitting solution for other scenarios.

VCDM heavily relies on a Local-as-View (LAV) modelling style without specifying the local views explicitly. On the client side, each tool holds data in its own model, but upon transmitting its data instances to a central bus system, the data is transformed into a single common (global) data model in the first step. After transformation, the data is stored into a central schemaless repository which holds a change history of the data as well as the actual instances, the "Engineering Data Base" (EDB) [77]. This storage can later be queried to visualize the data according to the common data model while sorting and structuring its instances in respect to each local model (by omitting non-common fields).

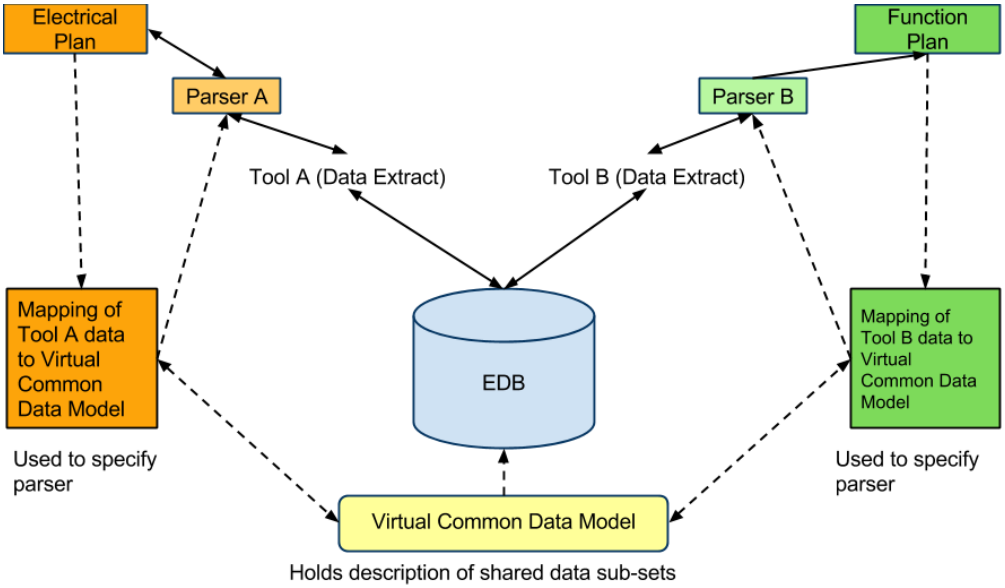


Figure 2.7: A Virtual Common Data Model mediates between two distinct tool data models

Service-Based Single Domain ("Classic ESB") Upon encountering the challenges imposed by the data modelling style of a VCDM, possible "counter designs" were also tested in the CDL-Flex during some of our integration projects. One of them brings the architecture more in the line with that of a classic Enterprise Service Bus [13] (serving as "backwards check" to see if a change of methods and paradigms is actually required): Each single functionality of the bus is encapsulated in a service and data is transferred along the services by a work-flow and turning the model explicit in a single domain. The work-flow encapsulates the instructions to send data from the tools along a pipeline of services before using a "Signal Domain" to store the data in the EDB. Illustrated services include a "Transformer Service" and an "Analyzer Service". The "Transformer Service" translates tool data into common data and vice versa, managing a set of transformation configurations. The "Analyzer Service" ensures data format validity (e.g. by inserting default values into fields that should not be empty or verifying the format of primary keys) and checks the new data for well-known errors (which do not violate the data model, but cause validity issues later on in the project). The "Signal Domain" ("signal" being the common data model in this example) is acting as an interface masking the actual storage system. Upon receiving new data, it creates a change set w.r.t. the current EDB state, which can either be shown to the user for merge resolution or silently applied to the EDB using a pre-defined merge behavior. If a user (or tool) requires new data from the bus, the signal domain provides the query interface to request data from the EDB, delivering instances from the common data model.

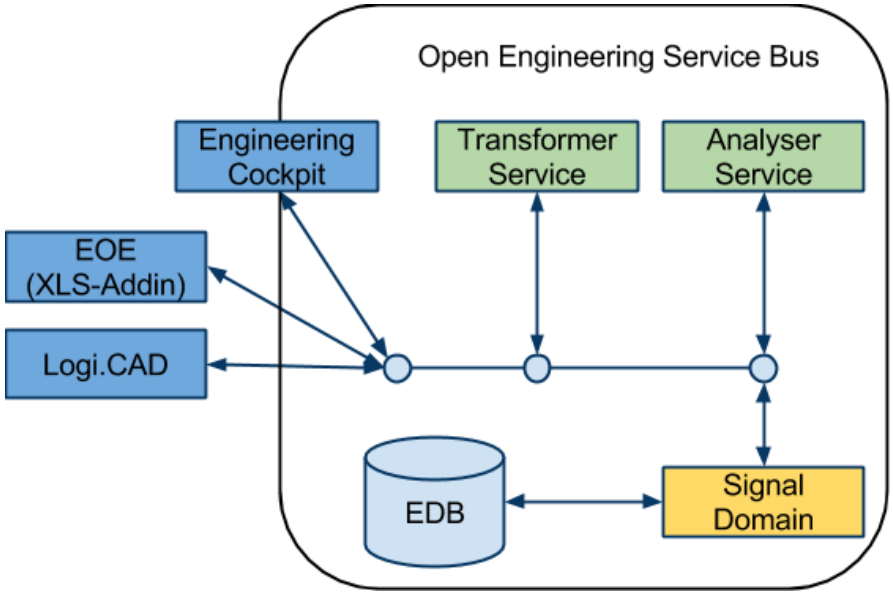


Figure 2.8: "Classic ESB" integration model

2.3 Architecture Evaluation and Quality Assurance

Below are short introductions to common procedures for the evaluation and/or quality improvement of software and design artifacts. First, as already hinted in chapter 1, testing a data model design or a specific architectural style by designing, implementing and evaluating the performance of the final "product" is rather inefficient. As this applies to software engineering in general, techniques for the initial evaluation of architectures already exist, one of which can be adopted for use in data model evaluation. Second, in software engineering, there are scenarios under which "testing" is hard to do, in our case the feasibility and efficiency of a data model, for which inspection provides a viable alternative.

2.3.1 ATAM

The Architecture Tradeoff Analysis Method [43] provides for a framework to analyze, validate and improve software architecture designs ("styles") against a given set of requirements and how to derive said requirements (referred to as "quality attribute characterization"). With offering a more holistic approach for architecture analysis, ATAM has been evaluated extensively [64] and been applied to a variety of fields [25] [62] [10] [59] [40].

Within the original technical report which also contains samples and templates for a more intuitive use, the core ATAM definition spans over 30 pages explaining its goals, wording and core process. Even without much detail about the goals and wording, the core process is quite easy to explain and understand, which is most likely to be one of the reasons for the acceptance of the ATAM. Said core process consists of nine steps separated into four stages:

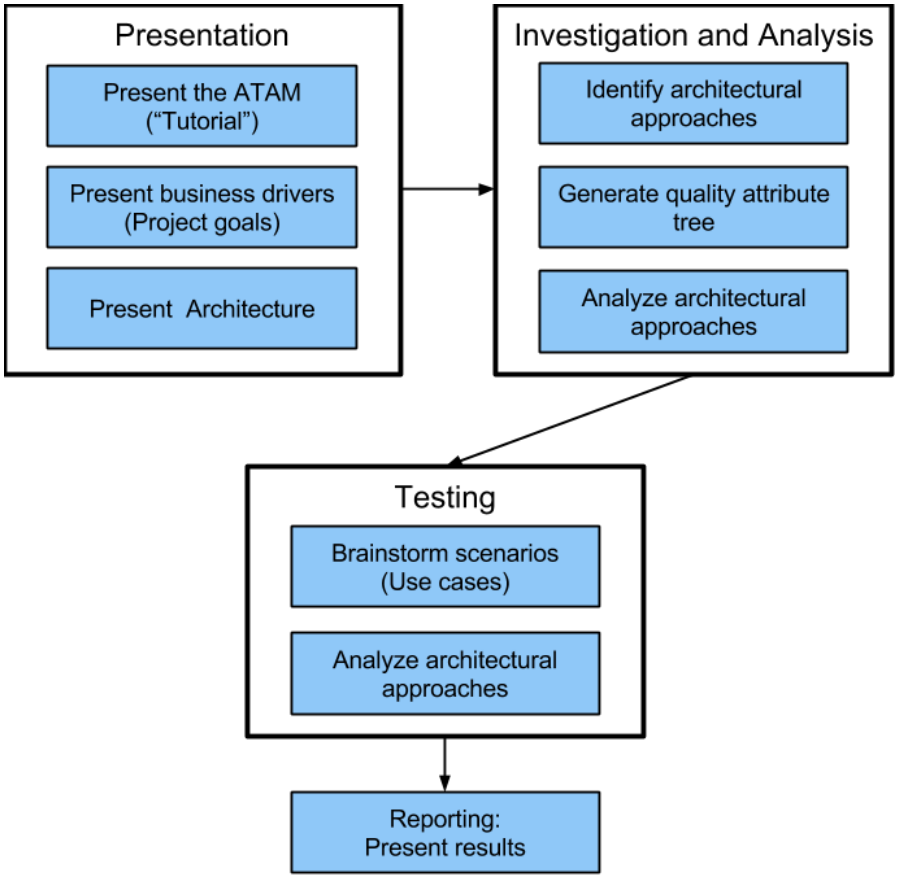


Figure 2.9: The nine steps of the ATAM

1. Presentation - consisting of steps 1-3 to present the ATAM itself, the business goals of the project and the current architecture

2. Investigation and Analysis - with steps 4-6 to analyze the architecture of the system under review, formulate current assumptions based on the business goals and evaluate the system under that knowledge

3. Testing - during steps 7 & 8 a larger set of stakeholders is invited to correct the business goal assumptions by comparing them to use cases and re-evaluating the system

4. Reporting - in step 9 to summarize and distribute the results of the ATAM execution

These nine steps are:

(1) Present the ATAM The ATAM is explained to all stakeholders (like customers, users, developers, system designers, managers, testers), ensuring that the goal of the process and its concepts/wording is understood.

(2) Present business drivers Usually the manager of the project explains what the business goals of the project are and what its main non-functional requirements are. Functional requirements can be used to illustrate the entire scenario, but the non-functional requirements will be the main source for later quality attribute characterizations.

(3) Present architecture The architect will present the currently planned architecture, trying to explain how it addresses the needs from the business goals.

(4) Identify architectural approaches The architect (and maybe other members of the team) try to identify used patterns and styles (if not already known) and how they relate to the business goals.

(5) Generate quality attribute utility tree The development team tries to "rank" the non-functional requirements derived from the business goals and to express them as quality attribute characterizations. Such QACs consist of three parts named "external stimuli", "architectural decisions" and "responses" which roughly correspond to (inter)action with the system, the style/pattern to deal with a scenario and the measurement for success in satisfying the quality.

(6) Analyze architectural approaches Based upon the most important QACs from Step 5, the "architectural decisions" that aim to satisfy them are analyzed in regard to their effectiveness. In this step the so called "architectural (non)risks" (whether a certain decision is "good" or "bad"), "sensitivity points" (points or outputs of the system that relate to the requirements), and "tradeoff points" (points or outputs where multiple requirements clash) are identified.

(7) Brainstorm and prioritize scenarios Apart from the initial ATAM presentation, this is the first time other stake-holders are brought into the process. By collecting "scenarios" (use cases and future change requests to the system) the other stake-holders are used to refine the initial requirements and provide them with a viewpoint to map their goals onto architectural decisions (and as a consequence, on QACs). In addition, by connecting QACs and scenarios, any previous specification errors or misinterpretations (from Steps 2 to 6) can be corrected. Finally, the top scenarios for further analysis are voted for. This way, using ATAM does not guarantee that the entire architecture is reviewed, but the parts most important for its business success.

(8) Analyze architectural approaches Under the knowledge of Step 7, Step 6 is repeated. Scenarios voted for in Step 7 are applied as test cases for further analysis of the architectural decisions which have been documented and measured so far. This may lead to new architectural decisions, (non)risks, sensitivity and tradeoff points being highlighted, (in)validated and documented.

(9) Present results Based on the collected results of applying the ATAM, these findings are presented to all stakeholders. Depending on the amount of corrections needed and the level of detail the entire documentation already has, a final report and recommendations for averting identified risks will be provided as well.

2.3.2 Inspection

Since a design, especially a data model, cannot be tested before implementation, other ways of validating a design have to be applied. In software engineering, reviewing code, documentation or design artifacts can be done in a formal way, using the procedures and best practices established from (software) inspection [23] [44]. This subsection does not aim to provide a complete overview on current inspection techniques and their modifications, but a short introduction into its core process and variants used in this work.

The Fagan Inspection [22] is often stated as the first officially defined method for software inspection, as it provides a standardized review process for (code) artifacts produced during a software engineering project.

Roles

In the basic (recommended) configuration, an inspection team for a software review, consists of four people: Moderator, Designer, Coder and Tester.

The Moderator is not directly involved in any other work concerning the artifact, organizes the review process (and its central meeting), collects the results of the review and distributes the results. This person should also check that the artifacts entering and leaving a review cycle satisfy a certain quality level, so that the review is not hindered by issues like spell-checking.

The Designer is the person who initially planned and designed the code under review.

The Coder is the person who actually wrote the code under review. If Designer and Coder happen to be the same person, the "Designer" role should be taken by that person, while the "Coder" role should be staffed with another developer.

The Tester is the one responsible for current or later testing of the produced code.

Stages

The review cycle following Fagan's recommendation looks as in figure 2.10.

Planning Not initially mentioned in the process description, but nevertheless important, the moderator plans the initial meeting of the inspection team and further sessions.

Overview The designer explains the requirements and plans for the code to review, in detail. This may involve elaborating the inner logic of the code artifact as well as explaining system-specific constraints.

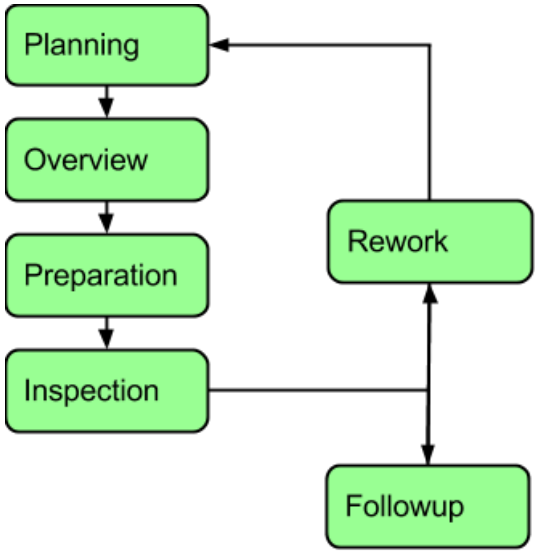


Figure 2.10: Fagan Inspection

Preparation Each team member reads the code and required documentations by themself to understand its design and structure. Really obvious and severe errors may be discovered here, but this is not the main goal of this phase.

Inspection The inspection team meets and goes through the artifact line by line, led by a depicted reader (usually the coder). Any errors discovered during this read-through are documented by the moderator. If a fix to the error found is obvious, it is noted as well, otherwise the reading will continue. After the end of the inspection meeting, the moderator distributes the results of the inspection in a written report to all participants of the meeting.

Rework The coder now works to fix all detected errors. After this phase is finished, the moderator will either start a new inspection cycle or continue to the "Follow-up" stage.

Follow-Up If all defects found in the inspection meeting are fixed and the verification of these fixes is trivial, the moderator can end the inspection cycle by him/herself, otherwise a new iteration will be started.

Notable variants of Fagan Inspection

After Fagan's method has been widely accepted, a variety of modifications have been created and put into use in the industry over the last years [4] [71] [45] [57]. Although Fagan Inspection can be applied without modification to review most artifacts produced during a software engineering project (not just code) [42], we propose some adoptions when validating data models for an integration solution.

In “State-of-the-art: Software Inspections after 25 years” [4], six notable alternatives and variants of Fagan Inspection are shown, two of which will be used to derive a review process: “Two-Person inspection” and “Inspection without meeting”.

Two-Person inspection Using this approach, the creator of the artifact calls for a review of his/her work by another member of the team who executes the review using a specific set of questions and a fixed schedule. This reduces the time and effort needed to setup a review cycle by a great amount while retaining much of its efficiency, as it has been shown that during a regular inspection meeting, participants often fail to pay attention or cannot make their point during a lively discussion/review session. In addition to the reduced demand in manpower, this approach also addresses the possible issue that a team of application integration developers may not have enough experts in data modeling who can participate in such a review session. As data modeling is a small subset of the tasks that need to be done during an application integration project, not all of the developers will have the skills required to properly review such a data model.

Inspection without meeting Although the initial modification for an inspection process may consist of reducing its participants, it may be advisable to allow others to join the process, especially people who will use the integration solution later on can be addressed this way. As an integration project may be implemented by a different group then the one who will use the solution, potential key users are sometimes not available for Q&A sessions to allow for clarification of the initial requirements or to double-check if the integration project is going the right way. Supporting these people with a method of validating the designed data model can yield high-quality feedback (as they know if the data models suit their work-flows or not) in a short time. However, as external participants cannot be expected to be present for a review session, it has to be possible to give feedback via a report or any other kind of asynchronous communication.

CHAPTER 3

Research Issues

The specific issues this work deals with are explained in more detail in the following 3 subsections.

3.1 Modeling Data Integration Solutions

"Which modeling concept is needed to allow for seamless data integration in a heterogeneous software landscape?"

Currently, development steps in projects of the automation industry involve several engineering disciplines and their (often) isolated software tools. Most of these tools were designed for a specific phase during development and therefore only provide a limited view of the artifacts engineers work on. Out of the necessity for the different engineering groups to work together, the groups agree on some common data model for specification and progress/artifact exchange. Such a common model is most often closely based on the data model of one specific tool or upon the actual physical structure of the object which is designed and built. For example, a software engineer talking about a pump may actually refer to the code they have written to control that pump, while the electrical engineer refers to the hardware and sensors connected to this pump which executes the control code and provides the inputs for the same software. "Only" the mechanical engineer may refer to the actual pump, but will also know what their colleagues are talking about. On the other hand, each of the three engineers is restricted in their activities by the work of their colleagues; the software engineer has to know the hardware specification (e.g. memory capacity and processor speed) against to program and what inputs will be available, the electrical engineer must know what range the sensors need to cover and the mechanical engineer needs to make sure that the components they put in place can be attached to the needed hardware and supports a series of specified behaviors (which the software engineer had to implement). Regarding this example (and the general scenario of collaborating engineering disciplines), it becomes clear that each discipline is only interested in a very small aspect of the overall data, but needs to know about the structure of the entire data and whether

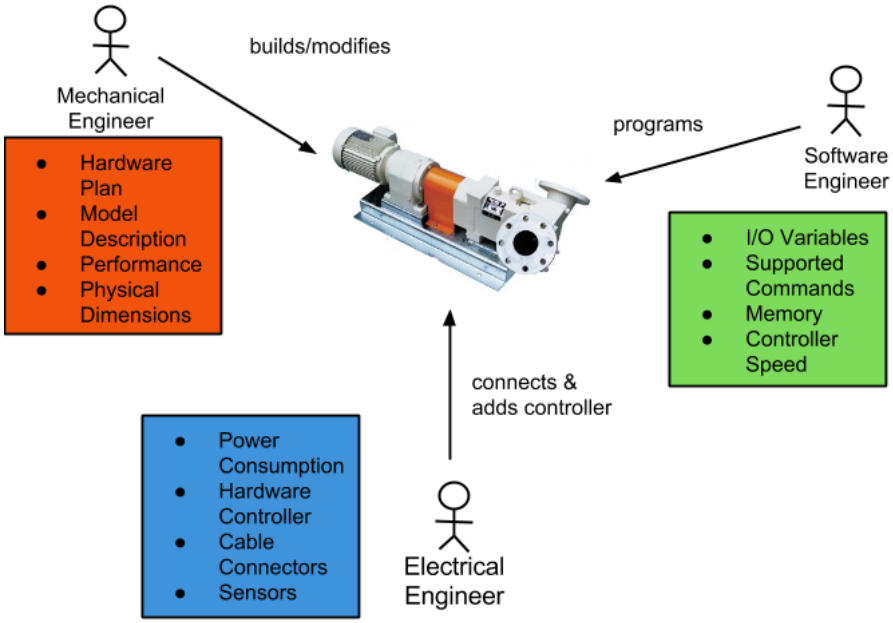


Figure 3.1: Distinct perspectives of rngineering disciplines

the current work of other collaborators affects their own. However, with a variety of proprietary tools used in this sector and the varying engineering workflows of the companies, there is not "the" one data model to cover all scenarios, even if limited to a very specific subset (e.g. the design of hydro-power stations).

From the observed diversity of tools, data models and workflows, it was assumed that a concrete data model has to be designed for each company and/or group anew, but must cover the following aspects (regarding engineering artifacts):

- Tool Data Models (TDMs): covering proprietary models of the tools in place as best as possible
- Tool Domain Data Models (TDDMs): abstracting from the tools how engineers of one discipline view their artifacts
- Common Data Models (CDMs): the view that is used between disciplines to collaborate, and that may be addressed directly, "outside" of the tools already in place

Apart from the need to cover these three views to interact with the current version of the engineering artifacts, there are other groups who wish to have access to some of the information related to theaw artifacts as well, the most common groups being quality assurance and management. Both groups are not interested in the details of the engineers, but to make quantitative checks on the data base on a regular basis. For example, quality assurance needs to verify that artifacts are properly reviewed and/or be able to match produced artifacts to specifications, safety guidelines and other criteria. The managers, on a much higher level of abstraction, need to be

able to perform proper claim management (i.e. see how many changes/regressions a customer causes mid-project by specification changes) and want to track the overall project progress. These combined requirements call for a data integration solution which enables all collaborating (and supervising) groups to access and modify current and historic artifacts from a series of different perspectives, which is summarized in the question opening this subsection.

3.2 Validating Integration Solutions

"After modeling an integration solution for a specific use case, is there a generic way to validate its designed data model (and mappings)?"

Designing a data and tool integration solution for interdisciplinary engineering workflows starts with defining (or collecting) the main workflows of the major disciplines involved. Once these workflows are known by the integration developers, a set of core use cases can be extracted. Those usually consist of the separate steps from said workflows together with a series of automated background tasks to promote their results to other participants. Before finally implementing the use cases, a complete data model is required to manually "simulate" the given workflows, such that the integration developers can determine whether they now understand them and are actually able to implement them. If the developer(s) feel confident that they have understood the specifications, a prototype (one or two iterations in the development cycle per use case) providing said features will be delivered, so that the users can (dis)approve the design of the solution. However, with the complexity of the workflows to reproduce and the fact that there are at least three layers of data models (see above), there is not "the one" best-practice-approach for modeling this system. This, in combination with the relatively late validation of the data model design may cause an erroneous design to "live" quite long before being fixed.

To make matters worse, different integration developers may depict different overall data models for the same task: While the tool data models are predefined by third party tools, the discipline data models used for connecting these third party tools to the integration solution and the common data models used for the integration solution's internal logic are not. Both of the two later ones are entirely free to be designed by the integration developer(s), often to compensate for shortcomings of the third party tools and to allow for usages not even remotely in the scope of said tools, such as refactoring, versioning or data mining.

From those desires, it becomes quite clear that the chosen modeling approach will have a large impact on certain models of the integration solution and even on how it will be used, as a certain "global perspective" of the data will shape the users' perception of their work. On the negative side, this also implies that if the designed data models are deficient, the consequences will be serious. First, adding "non-user" features (features not directly derived from existing workflows or functionalities of the third party tools), such as the already mentioned refactoring, versioning or data mining capabilities, may result in high effort. This can happen quite easily if the integration developer modeled the discipline and the common data model too closely after the tool data model, reproducing the shortcomings which the integration solution should overcome. Second, if the discipline and the common data model's representation to the user differs too

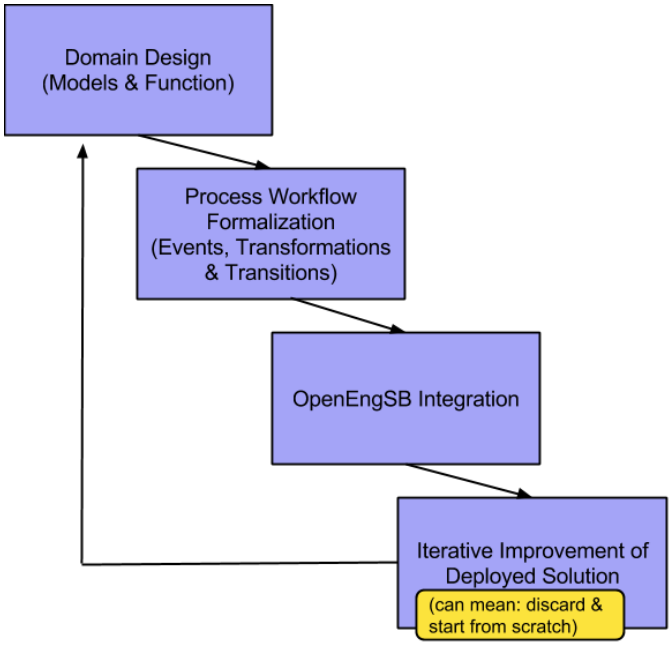


Figure 3.2: (Simplified) development process

much from the users' known data model, they may not understand them or fail to recognize their workflows "in" the integration solution.

In previous prototyping approaches, we used a basic implementation of the integration solution and let users (manually) test it, to see whether they can perform their use cases as intended. With a growing number of use cases and a set of already established (partial) models, this approach became more and more ineffective, as the review cycles got longer and use cases started to become interdependent. Quality assurance in terms of data model design should reduce the amount of faulty model drafts that "reach" the end-user. Certain types of defects, such as broken workflows due to impossible data transformations can even be detected by developers who are not familiar with the target domain. Upon review of previously encountered design defects, we aim to produce a series of "generic" model design checks which can be performed before implementing such erroneous designs. After such a series of checks is established, an efficient method for performing them has to be found, whereas we will focus on the evaluation of methods based on inspection techniques. Using this approach, we try to reduce the initial implementation effort by relying on an inspection of the data model without any previous implementation. This is done by applying a checklist-like procedure on the design before-hand. Evaluating if this "dry-testing" of design enables an earlier detection of known design pitfalls and a shorter review cycle for designing said models leads to the second research question, stated at the beginning of this subsection.

3.3 Modifying Existing Integrations Solutions

"With reoccurring changes to the data models, how can we distinguish between valid changes and erroneous modifications?"

Once a general architecture for data integration in a specific field (such as the construction of steel-works) is found, another challenge will become immanent: The data models used for integration and the data models to integrate will change over time, even within a specific group or company. This has several reasons.

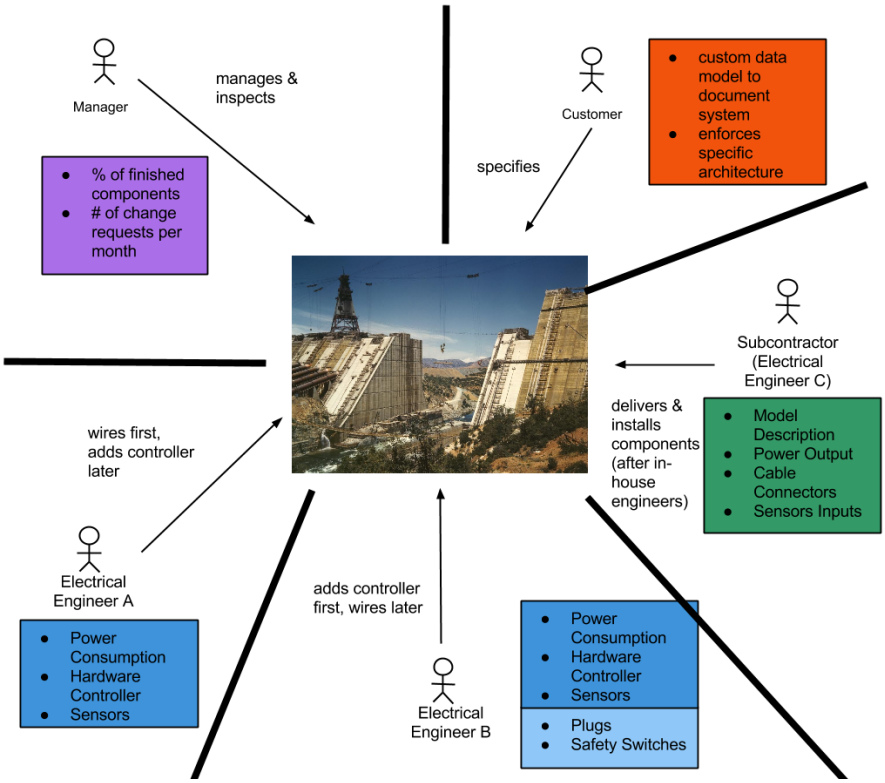


Figure 3.3: Stakeholders that influence a common data model (among other things).

First, our observations in the automation industry have shown that engineers of the same discipline may have very distinct workflows and modeling approaches. This is due to the fact that people are specialized in the design and implementation of different facilities (e.g. an assembly with a single large belt versus a factory with a series of small belts and complex interlocking controls) and have adopted to whatever style works best for their field of expertise. Under these circumstances, each (new) engineer may not only introduce slight differences in the importance of parts of a data model, but also introduce new fields (or discard some) for their personal "workflows" (which this person is honed for).

Second, the costumers themselves may induce a specific architectural style which can have a severe impact on the project, if it does not match the "standard" model and workflow of the

company which runs the project. To illustrate why this is possible, one has to understand that, unlike in "pure" software engineering projects, the final product ("the facility") is not a black box to the customer, but an open system which he both interacts with and modifies as it is used. Also, the interaction, unlike with end-user software, is not completely abstracted via a (user) interface, but happens (to some degree) on a low-level basis with the customers understanding the internals of the facility (mechanical components, electrical wirings and its software). So, it becomes clear that the customer may want the inner structure of the final product to be adjusted according to their needs much more than in software engineering projects.

Third, the managers may cause changes to the workflow and data model. Depending on the contract set up with the customer, they require different ways to track a project's progress (from the above example, "What is the status of the single belt's wiring?" or "How many belts are finished?") and/or determine the customer's influence on a running project (e.g. the number of change requests on components in development due to unclear/wrong specifications). Retrieving such information from a running project requires to document changes made by both engineers and customers and to establish a valid "benchmark" for the project's status which is linked to the actual data. In more extreme cases, the need for making progress data "visible" can cause engineers to be forced to reverse parts of their workflows (e.g. write software for hardware not yet designed or build) which in turn leads to data model changes, as many previously valid assumptions and shortcuts do not work. For example, if a series of input and output parameters for a control program were auto-generated (or copied) from circuit-board plans, the software engineer now needs to create, manage and store a set of data they do not "control" usually.

Fourth, there is the element of subcontractors. The cooperation of companies to build a facility often results in a mix of hardware and software from different vendors. Sometimes, these additions are adapted to the "main" system by hand, as if they were made by the prime contractor itself, but in other cases, the sub-system is treated as an external facility and some kind of (individual) abstraction layer is implemented to integrate the sub-system. These abstractions usually introduce an entire new set of data models into an already complex design.

With as many possible sources of change, it is not uncommon for the integrated data model to be changed mid-project, as some of the changes implied above do not pose an issue until a certain phase of the project (or the simulation of reaching this point). However, on updating the data model, one needs to be aware of the fact that some proposed changes may not be useful or necessary. Sometimes an engineer (or a manager) overestimates the degree of deviation of their local data model compared to the tool, discipline or common data model and tries to adapt higher-level data models according to their needs for no good reason (e.g. if an engineer uses a series of fields for their local data models, but no-one else requires or effects them, it will not be necessary to mirror them in the discipline or common data model). Dealing with regular change requests for the reasons mentioned above motivated the third research issue, which started this subsection.

CHAPTER 4

Methodology

This chapter details on the method used in the following chapters. First, the general approach to find and evaluate a possible solution is given. Second, an example scenario for a multi-disciplinary engineering work-flow using heterogeneous software is described which will serve as our later "main use case" to test and refine our solution.

4.1 Approach to Derive a Good Solution

Finding a solution to the questions stated before consists of four steps: Initial literature research, prototyping, evaluating the design for a given scenario and creating a guideline to validate instances/updates of the design. Any shortcomings found during the evaluation will be used to improve the initial model and to draft a change-request-handling procedure for the data model design.

The literature research (see previous section) aimed to find out about currently known best practices for data integration in large software systems, common standards in the automation industry and evaluation methods for architectural styles. Further fields of interest will be quality assurance methods for data model designs and current methods in Enterprise Application Integration.

In the second step (detailed in the next chapter), iteratively developed research prototypes will be used to find a data design model pattern for a specific cross-discipline engineering work-flow. Using this scenario, previously known modeling styles, based on current best practices and amendments from the CDL-Flex, the advantages and disadvantages of multiple approaches in the same scenario are gathered. From the knowledge gained through the prototyping phase, a specific modeling style is extrapolated and compared against the already applied styles in terms of robustness against changes, efficient integration of third party data models and "openess" to allow for new features. This step follows the development cycles and maturity levels detailed in "Research Prototypes versus Products: Lessons Learned from Software Development Processes in Research Projects" [19], however the prototypes developed or planned for this work only range between maturity levels 2 ("Research Concept") to 4 ("Quality Assured Prototype"), as

categorized in [19]. (Since the initial research to come up with the basic scenarios and concepts, as well as the "finalization" of prototypes, are beyond the scope of this work.)

In the third step, the architecture derived from the initial "fast prototyping" is evaluated using the Architecture Trade-off Analysis Method (ATAM) to check whether the ongoing developments actually serve the needs of a representative application integration project and identify overlooked requirements (in hope of being able to modify the architectural approach accordingly).

Finally, we illustrate the creation and reasoning behind a step-by-step modeling guideline to create a specific design and a respective method how to validate it. This is done by identifying known mistakes, pitfalls and risks for each step, formulating counter-measures to these issues and creating questions that indicate whether an issue has occurred or not and providing means to react to it. The same procedure is applied to deal with changes to existing models, as our "testing" scenario includes mid-project changes and extensions to the data model as well.

4.2 Evaluation Scenario

The "standard" use case for this work is the review process in a multi-disciplinary engineering project, called "Signal Deletion with Review". This scenario involves three engineers, one from electrical engineering and two software engineers (although the specific disciplines do not matter).

The Electrical Engineer works on the Circuit Design and uses the tool EPlan[1] for the project. The software engineers (A, B) use logi.CAD[2] for control software programming. Those two programs use different data models to store their data, but they are mapped via a common data model, namely the "signals" (referring to the I/O-pins of the hardware or the respective variables in the software). The structure of these signals is a simple list of key-value pairs. The transformation from the original representation of the values to the common data model is either provided by the program itself, a script or by a connector that links the program to the information system. As the engineers collaborate in a project, this means that their individual work can affect the work of others, even if they are not changing someone else's data directly. At the start of the sample scenario, the status of the information system is not empty and consistent, i.e. some previous work has already been done in the project. Some signals already were created, some of which refer to actual hardware and/or software variables, some are not finished yet and others are blank templates created for "later use", e.g. as spare inputs for later repairs or extensions.

Now, the eletrical engineer (EE) modifies some local wire circuit design, e.g. having figured out how to solve a certain wiring issue. This change made within the EPlan software then demands a propagation of modified data to the integration solution, which triggers the review process. If the modification done by EE does not cause any signal deletions (that is, all changes affect the local data model only), nothing else happens. Otherwise, A and B, as being from a different discipline then EE, receive a notification about the changes. (As a side note: Current collaboration solutions that work within each discipline exist). On inspecting the changes to their

[1] `http://www.eplan.us`

[2] `http://www.logicals.com/products/logi.CAD/`

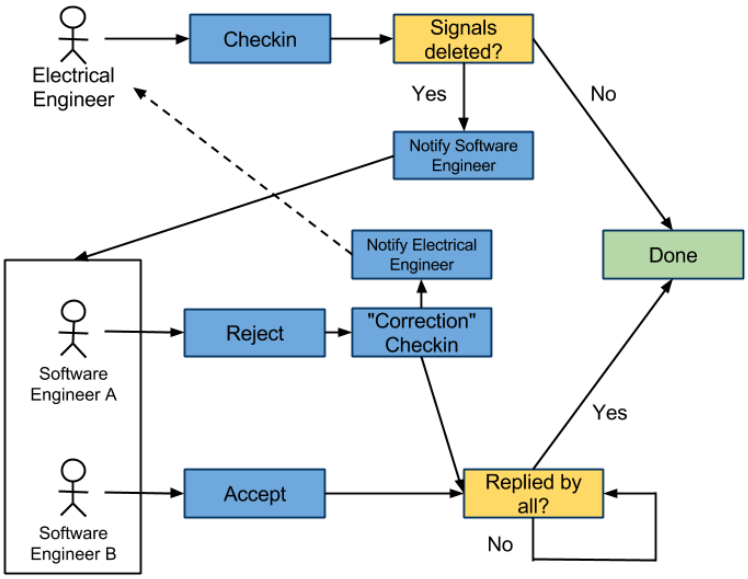


Figure 4.1: Signal Engineering Review Circle

local data, the software engineers now need to make a decision: Accept or deny the changes. In the first case, A rejects the change, as e.g. it removes some variables which are known to be required later on. As a simple "No" is quite unsatisfactory (this would stop further progress), A now has to provide a "Correction Check-in". This means that after applying the (undesired) changes locally, A introduces the (now) missing variables again to produce a change-set fixing their local issue. This change-set can be seen by the EE as well and could potentially trigger a new review cycle, however this is treated as a new instance of the process. After the check-in, the vote of A is now finished and the review will be done once all other participants (in this case: B) have submitted a vote (or a timeout has expired). B then simply accepts the change (most likely aware of A's adoptions) and "unlocks" the final change-set (changes of EE with fixes of A) for their continuing work.

CHAPTER 5

Data Model Candidates

This chapter details on how the suggested data model for evaluation in chapter 6 was found. First, we illustrate the development process of the initial candidate for a data integration style and its evaluation by prototyping and user testing. Second, the re-design of said approach to come up with the final solution candidate is explained to highlight the issues which lead to the suggested style.

This (first) part deals with the design, testing and inspection of four different architectural styles which each promoting another modelling style. These approaches have been implemented to a varying degree, either for testing under real-world conditions or until a fundamental problem with the respective data model was discovered.

5.1 Virtual Common Data Model (VCDM)

VCDM-based prototypes were already deployed at two of our industry partners and had undergone a series of user test iterations, reaching the maturity levels [19] of “Quality-Assured Prototype” (4), but failing to be elevated to “Industry Product” (5), for a series of known issues that seemed to be inherent results of the design and could not be simply fixed be minor adoptions. So, with VCDM being the used modeling style in many of the CDL-Flex research prototypes, these ongoing issues and seemingly increased development effort called for a re-evaluation of this style. While allowing for a quick way to integrate several data models from various disciplines and a performant change management system, it became clear that this approach has several weak points.

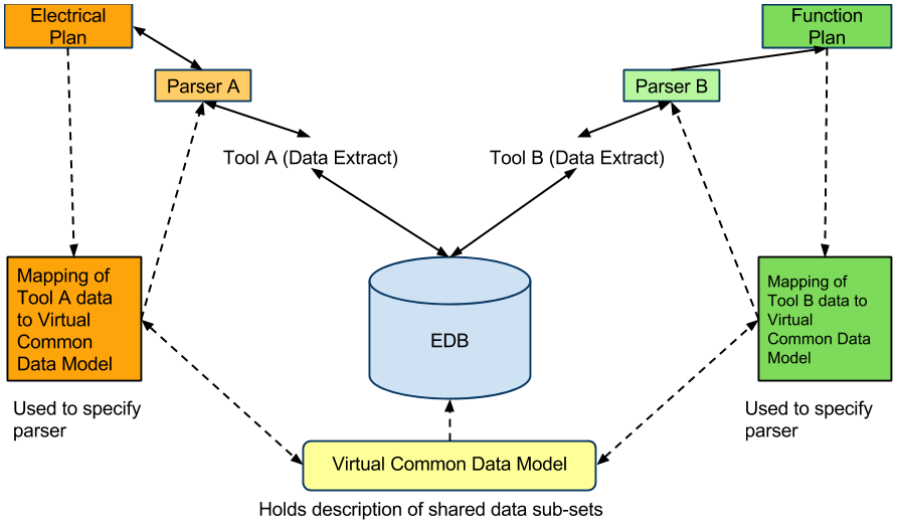


Figure 5.1: A Virtual Common Data Model mediates between two distinct tool data models

Limitations First, without any documentation or other explicit representation of the VCDM (as there is no actual database schema or class that resembles the VCDM), application developers require external documentation of the global data model and its transformations, as there is no "true" occurrence in either the source code or the data base configuration which can lead to a variety of errors and/or false assumptions during development. Second, the lack of an actual global data schema in the storage solution (which allows for easier integration of new data models) demands for schema-enforcement in the software used to push the data into the EDB. This enforces either the use of a constant data model in the top layer of the application, interlocking software and data model across several layers or the use of a highly generic data model, which creates lots of boiler-plate code on the front-end level. In both cases, changes to the data model cause code refactorings in a multitude of places, making the code hard to maintain. As long as the frequency of models changes remains low (i.e. there is a one-time integration of a huge system) or the bus application itself does provide little tool-domain-specific functionality, this is not much of an issue. However, with unstable global models or frequent refinements of the specifications of local tool data models (as described in section 3.3), VCDM might not be a good solution. Third, using this configuration, only one common data model can be efficiently addressed and managed by the tools, making it quite burdensome to integrate more then one discipline-spanning data model.

5.2 Service-Based Single Domain ("ESB-like")

In view of the apparent issues of the VCDM, it was decided to re-view the classic Enterprise Service Bus concept, at least on the level of a "Research Concept" by trying to design integration solutions for our use cases. Since previous research efforts, such as [60] indicated that the ESB is unlikely to provide for a satisfiable solution, the evaluation was limited to the design of an

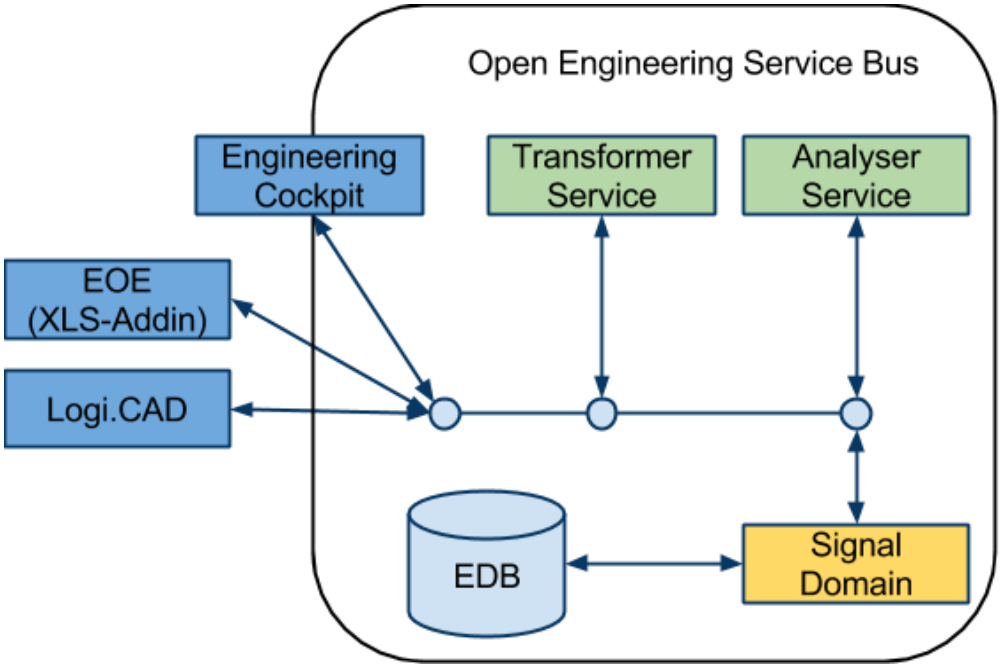


Figure 5.2: "Classic ESB" integration model

ESB-based architecture.

Unlike the VCDM style, the approach makes the data model explicit, allowing for easier testing and refactoring of individual components, also allowing new developers to understand the common data model much easier. In addition, by using a "service pipeline", new processing steps (for new features) can easily be added without having to change existing workflow from the user's point of view. A special case of "adding a feature" may even include to skip using a specific service if it is not needed under some conditions (e.g. bypassing checks at some time in a project).

Limitations The obvious weakness of this design lies in the fact that changing the data model still causes changes in all services along a workflow, as there is no central authority governing the data models inside the bus system. Although these changes are now fairly easy to perform, as every service is a smaller, less complex component then before, failure to adapt all components can yield unexpected results which are not detected at first glance. An example for such a mistake was that if the meaning of a field in a tool's data model changed (whether because it was defined wrong in the first place or its use in the actual tool), which in turn changed the mapping between the tool data field and its respective field(s) in the common data model. As long as any working transformation and data analysis is present, a half-executed data-model change can go unnoticed.

A lesser issue with this style is that it fails to utilize the advantages of the Engineering Service Bus by reducing the tools to mere data sources and sinks (like in the initial prototypes).

This is fine for initial demonstrations, but on the long run it means that any features of the local tools need to be either re-implemented on the bus or be brought into the bus by some other means, ignoring one of its core concepts, the domains. Also, the tool data models now are somewhat obfuscated, merely shifting one of the core issues with the VCDM-style from one are (the common data model) to another (the local data models).

5.3 Everything is a Domain ("ESB-Domains")

During the implementation and validation of the second ESB-like prototype, it became clear that a central authority for data models across workflows is needed and that it would be convenient not to create a service-based infrastructure from scratch (as the Engineering Service Bus aims to ease the use of external and internal services via domains). Under these assumptions, an alternate design style was proposed: Turning every service into a domain and turning the common data model into the "gate keeper" of the bus. Following these ideas, the "Signal Domain" holding the common data model was turned into a tool's entry point to the bus and the sole component it would interact with directly. In addition, all further interaction between the user and the EngSB is abstracted into separate domains, one for each type of interaction (e.g. "Notification" and "Review"). Also, most internal services are wrapped into domains so that workflows can easily address them during runtime, having all features deployed on a running instance readily available. This approach was tested by implementing a small research prototype (maturity level 3) that was only tested inside the CDL-Flex, but not at our industry partners, using known use cases and previously encountered issues.

Choosing this method promises three major advantages: Smaller components, robustness against data model changes and simpler workflows.

Smaller components With each service of the bus wrapped into a domain, much boiler-plate code for setup and data transfer is delegated to the generic domain implementation, therefore reducing code duplication in the services. In addition to making services smaller and easier to maintain as they now only contain their "functional" code, it also makes them more similar in terms of use, as the generic domain implementation and interface enforce one global style for all services. Another unexpected positive side-effect of this refactorings is that now one easily sees what a specific bus instance is capable of by simply listing all domains and their methods.

Robustness Against Data Model Changes By making a domain (which by itself is little more then a generic interface with an attached model) the entry point third party applications have to use for interacting with the bus, we aimed to catch issues with changing data models early on. If the specific model of one tool changed, simply adapting the tool's connector (which can be simplified as an implementation of the domain) suffices most of the time, causing no changes inside the bus. Even if the domain data model changes, it does not affect the bus in many cases, as the data model of a domain is never used directly, only via generic methods unaware of the actual implementation and its specific model. So the only case in which a model change affects the bus is if one more fields that were explicitly addressed for workflow rules or inside an internal

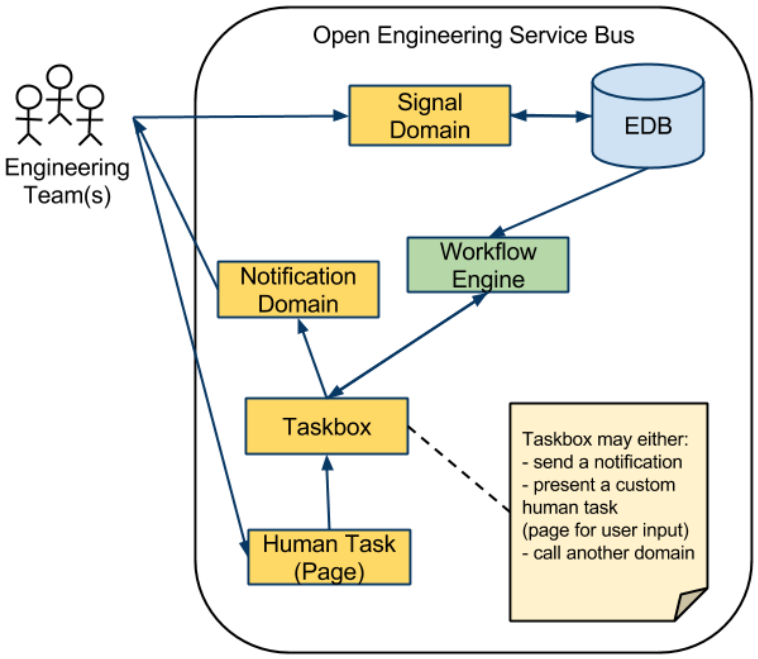


Figure 5.3: Turning all (even internal) services into Domains

domain changed (e.g. the notification would include parts of a changed signal into its message to the user).

Simpler Work-flows As all services deployed into the EngSB are now present as domains, their use in workflows is standardized, meaning that configuring workflows now is an easier task. Every step of a workflow now only consists of invoking a certain domain, collecting their respective result (“event“), dissecting it to get the correct parameters and passing them to the next domain.

Limitations While addressing most of the current challenges we faced with tool and data integration, not all of them are solved and some are not addressed in a satisfying manner.

First, if only the common data model is kept inside the bus, most of the advantages of having a versioned storage are lost. The main motivations behind versioning are (short-term) branching and reverting of older revisions. Branching in our cases refers to the possibility to keep several different copies of the current revision during a merge/review process (which can gon on for some time). This is necessary so that engineers can continue their work during such long-running processes without having to keep local (unmanaged) copies of their work. Reverting was designed to help retrieving a clean state of work more easily in case some change-sets prove to be erroneous (e.g. break somebody else’s previous work). In both cases, after changing to a given revision, each local tool somehow needs to restore the values of all non-common fields (i.e. not present in the domain data model). A workaround to this problem is storing all data in

the domain by declaring ”additional“ fields which are not present in the domain’s data model, but which can be accessed if their name is known. However, in both cases, adoptions to the tool data model (and corresponding transformations to get tool data into a domain) remain poorly manageable.

Second, by providing a single domain model, the bus enforces some kind of standard onto the companies using it, even if that model can easily be written from scratch for each company (which happened). In addition, by keeping the tool data models ”outside the bus“, neither the workflow nor any of the bus’ internal services could exploit distinct features of the connected tools or provide any functionality based around tool-specific data.

Third, after having deployed three different prototypes at different research partners, it became clear that several ”standard“ service were required all the time and in almost the same way, which raised questions whether such ”core functions“ should really be implemented as exchangeable domains.

Fourth, connecting all third party tools via a single domain brought up an issue which was not fully clear before. Sometimes, tools are either data sources or sinks but not both and have a certain hierarchy, i.e. one tool’s changes will always override another tool’s changes. If all tools’ input is merged together into a single domain at the very start of each workflow, it is quite hard to treat them differently and calls for solutions which are, at least partially, located outside the bus (and mostly result in hacks at uncommon places).

In short, a design that enforces a de-facto standard for tools’ data formats and promotes to solve challenges for the integration solution ”outside of it“ does not seem to be a good solution approach as well.

5.4 Multiple Domains and one Engineering Object (”EngObj“)

Facing the shortcomings of the initial approaches, the architectural design was modified step by step to single out those weaknesses, so that existing prototypes could be adapted should the need arise, i.e. if rewriting them and replacing them at industry partner’s side was unfeasible. In the first step, each tool group (which more or less resembles one engineering discipline) got an own ”tool domain“ inside the bus. The same ”tool domain“ would take the gate keeper role for that one discipline and relieve the ”common data domain“ from holding tool-specific data. By doing this, the former ”data domain“ was stripped of any affiliation to specific engineering tools, now being used for data validation and cross-discipline interactions only. To attribute this change, the data models held by this ”no-tool domain“ are referred to as ”Engineering Objects“ and the domain itself hidden from any (external) representation of the OpenEngSB to avoid confusion among existing users (the Engineering Objects will be displayed though).

As indicated above, the main purpose of this separation is to allow for a versioning of tool-specific data (or at least discipline-specific) without cluttering the inter-discipline data model and storage by doing so. This separation of the tool data versioning history combined with a duplication of the some of this data in the common model leads to a two-tiered change history: One for each change the users perform with their local tools and a higher-level change log in the common data model (e.g. if an engineer changes data in a field only present in the tool domain, but not in the corresponding ”Engineering Object”, change logs will look differently).

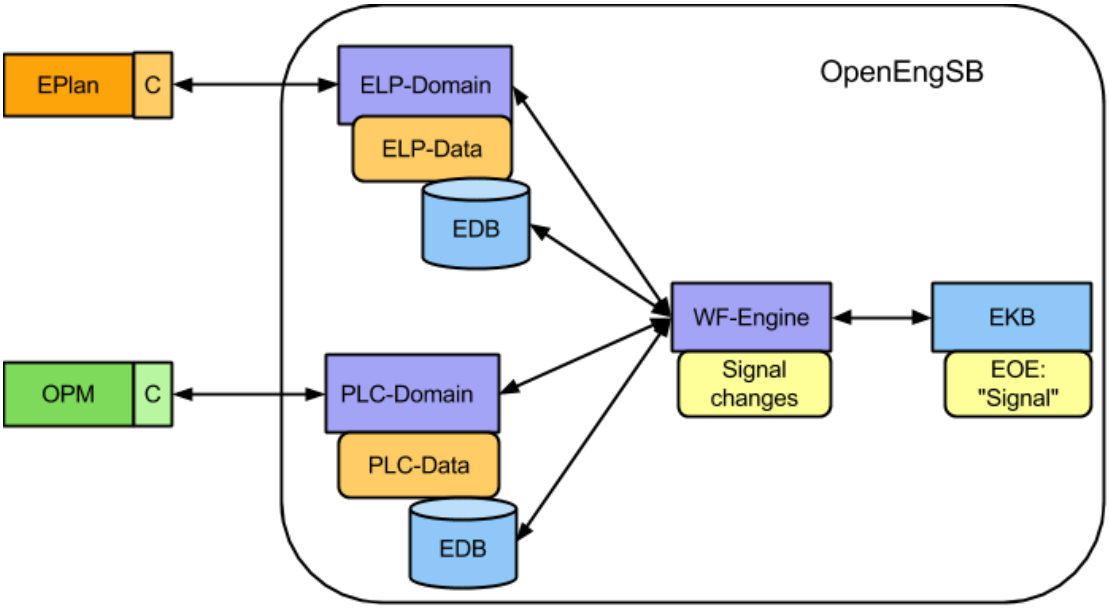


Figure 5.4: A separate domain for each discipline, one common model

This allows for an easier separation whether a specific change is of interest to other disciplines as well as for a more obvious change history along a tool-chain (“Which tool group/discipline triggered this change?”).

Furthermore, as a positive side-effect (as this was not considered beforehand) of this change, the nature of data conversions changed: They are now split into tool-to-domain (”mapping“) and domain-to-domain (”transformation“) transformations and performed by different services. The more basic ”mappings“ are still done by the previously used ”Transformation Service“ (using Smooks[1]). Such mappings mostly consider of simple field renamings and basic string operations (such as splitting, concatenating, extracting substrings and formatting) due to the fact that a tool’s data model and its corresponding domain’s data model are quite similar by design. The complex domain-to-domain transformations are handed over to the new ”EKBTransformations“ as the conversion of data between two different disciplines proves to be more challenging.[2] These transformations could be the same as the mappings mentioned before, but in some cases, more advanced computations are needed. For example, one case of connecting the electrical engineering tool and a software engineering tool require the computation of an address/variable field in the SE tool from two fields of the EE tool. This results in a two-dimensional look-up table followed by a computation of the address from the look-up results.

By separating these two conversion types, it becomes easier to allow power-users to deal with the rather simple and frequently changing configuration of tool-to-domain mappings (as the tool’s export formats are project dependent). Meanwhile, the integration application developers can focus on tackling the domain-to-domain transformations, which in most cases are the main

[1] `http://www.smooks.org/`
[2] `http://requirements.openengsb.org/confluence/display/MANUAL/semantic`

motivation for the entire integration solution, as they are more difficult to automate.

Along with establishing core services for mapping and transformation, a further set of ”standard“ features is put into the OpenEngSB framework (which previously were part of the specific integration projects). Business logic components executing the engineering use cases (i.e. committing and merging changes) are now managed as workflows[3], turning hard-coded sequences (as they were deployed as compiled classes) into exchangeable configuration. Another newly created service that fits well into this design, is the so-called ”QueryDB“, a database mirroring the versioning histories and allows for additional querying of the Engineering Objects. As a former by-product of the Engineering Cockpit [56], the QueryDB was initially build as a means of overcoming the weaknesses of the first EDB version, i.e. its extremely low performance when aggregating change logs and the complicated querying of specific changes (for one version). However, with the use of the workflow engine, command chains and corresponding events become a more present concept in the Bus, which in the end, justifies the use of a separate database for specialized queries and/or visualization, much like in CQRS [26] [80] [17].

Turning the Engineering Objects into the main data source for most further bus-specific activities and features, allowed us to experiment with other services, without breaking an already working integration project. One example are EDBHooks[4] for ”post-mortem data validation“, like as with Continuous Integration and Testing, which can be attached on any domain data storage without any interferences to existing workflows.

Limitations In short, one added layer of abstraction and separation allowed us to tackle the challenges facing from the previous approach. However, these benefits come at the price of higher complexity in many places.

First, the data model to design (tool domains and common data model) is much more complex then before and may pose an over-engineered solution if one only means to integrate one or two tools from a single discipline, thus limiting the scenarios when this design should be applied to larger application integration projects. This could also lead to rejection of this style if demonstrated on ”simple“ systems, as its benefits may not be clear from the beginning.

Second, considering the components used for mapping, transforming, storing (EDB) and processing (workflows) the data, this approach goes with a higher number of more or less isolated subsystems which are configured differently and therefore a higher chance of ”transitive“ errors. This refers to the situation that a missing/invalid configuration in one component, which has no immediate negative effect, causes unintuitive behavior in another component. For example, if at the mappings, a field which is later used for cross-domain reference is built in a wrong way, such that it points the wrong data instances. The transformations are likely to produce a ”working“ result, in the way that a data instance is referenced, but in the workflow later on, this causes the wrong referenced object to be queried. This in return may lead to an erroneous change propagation across domains which is quite hard to detect afterwards.

Third, this style is prone to the specific design/code smell of doing the right things with the wrong tool/in the wrong place, called *Feature Envy* [27]. Since both mappings and trans-

[3] `http://openengsb.org/manual/openengsb-manual/v2.0.2/html-single/openengsb-manual.html#user.workflows`

[4] `https://github.com/chhochreiner/EDBHook`

formations can do similar things, it can be hard to figure out which data conversions should be done where, if there are no guidelines or prior experience. Furthermore, as the workflow engine's (Drools[5]) rule set is a turing-complete language and can import and use classes from other (OpenEngSB) services, one can be tempted to include data validation, merge logic or any other feature present in the OpenEngSB simply as an extended rule instead of passing the responsibility for a specific task to the component designed for it.

However, none of the shortcomings of this approach actually threaten the feasibility of the data model design. All of the possible issue when applying this approach regard documentation, coding style and quality assurance concerns which we assume can be addressed by adhering to a defined set of design and validation set. In later chapters, we introduce measures that can help to reduce or eradicate the mentioned risks associated with this data modeling style.

5.5 Comparison of Modeling Styles

Summarizing the features of the data model design styles produces to the overview below. As mentioned above, this indicates that the "Multiple Domains and one Engineering Object"-style is our best candidate for a good solution.

	VCDM	ESB-like	ESB-Domains	EngObj
Robust Against Changes	no	no	yes	yes
Can Hold Multiple Common Models	no	yes	yes	yes
Versioning	yes	yes	no	yes
Service-Reuse	yes	no	no	yes
Only Valid Data	no	yes	yes	yes
Data Sovereignty	yes	yes	no	yes

[5] `http://www.jboss.org/drools/`

CHAPTER 6

Design Style Evaluation

This chapter illustrates the evaluation of a data modeling style and the drafting of guidelines to test specific data models and updates thereof. First, the new design approach is evaluated against our main scenario using the ATAM. Second using experience of known issues when modeling integration solutions in the CDL-Flex, a set of questions is derived to allow for a validation of the given data model. Third, changes to the given scenario that influence said data model design are listed to understand the volatility of (seemingly) stable data models. This set of volatilities is used in chapter 8 to introduce an updated evaluation guideline to provide for validation of changes to such data models during the life-cycle of an integration solution. (The same questions used in this validation are used, but modified in order and scale).

6.1 Architecture Evaluation with the ATAM

Before implementing a feature-completed prototype (for a feasibility evaluation) the architecture has to be reviewed once more, regarding the known (and often) poorly specified requirements. Most of the projects the CDL-Flex handles are either feasibility studies or side-projects alongside even larger integration projects, so its participants rarely know what to expect, which calls for a more formalized approach of evaluating a design. The initial steps of Architecture Trade-off Analysis Method (ATAM) allow for the internal review (by the developing team only) of the design, while later phases include other stakeholders, i.e. when more documentation and a better understanding of the architectural implications have been established. An intended positive side-effect of this procedure is that it most likely causes a critical reflection of the architecture from both perspectives (developers and ”customers“), as any previous discrepancies will come to light when discussing the architecture’s goals and limitations in the late phases of the evaluation. This is because the varying interests of the different stakeholders (and their different goals in the project) will cause deviating, even conflicting goals, which in turn allows for a better understanding of each stakeholder’s (actual) most important needs.

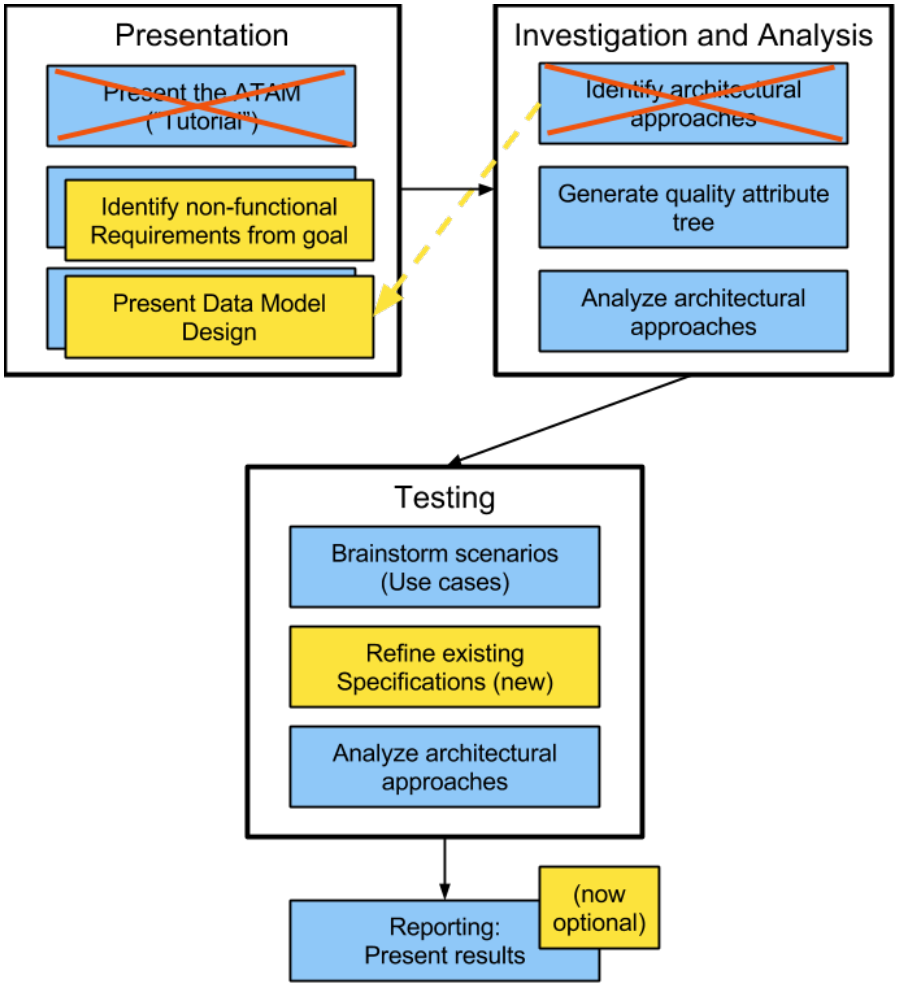


Figure 6.1: The modified ATAM

6.2 Extended/Adjusted ATAM Process Approach

For convenience and the limited availability of our industry project partners, the execution of the ATAM was modified to reduce the time effort for non-CDL-Flex participants and to make use of existing documentation. Furthermore, as the project chosen for this evaluation was already running, the time-frame of the ATAM was adapted from the (recommended) block of several days to a period of two months, allowing for two meetings at our industry partners' side.

The nine steps of the ATAM were changed as follows:

(1) Present the ATAM Only a few people were informed that the ATAM was conducted, as this was not the main goal of the integration project and likely to cause confusion among engineers and managers ("Why do we need such an evaluation for a prototype? "). Basically, step 1 was skipped.

(2) Present business drivers Since managers already did this in the past, existing minutes of meetings and knowledge of the project from members of the CDL-Flex team were used to identify the business goals and non-functional requirements. Also, the focus of this inspection is shifted even more in favor of the non-functional requirements, as the functional requirements, i.e. capabilities that can be tested more directly, are more likely to be connected to parts of the architecture other then the data model design (as the data model itself does not provide features, but enables or hinders them).

(3) Present architecture As with following steps, the architectural review is limited to the data model design style and therefore only the goals that can be covered by it. This restriction may lead to the conclusion that some of the (non)functional requirements are not addressed by the data model design, but by other parts of the architecture. So, for the scope of this work, all requirements that are aligned with the research issues need either be covered by the data model design (which after all is the core topic of this work) or need to be included into the review.

(4) Identify architectural approaches Step 4 was merged into step 3, as the chosen architecture is known at this point and the relation to the business goals is being identified in step 3.

(5) Generate quality attribute utility tree Step 5 is left unchanged regarding its way of execution, but limited onto the data model design, in contrast to the entire architecture. This, combined with the results from step 4, allows to identify which of the business goals should be addressed via the data model design and which actually are covered this way.

(6) Analyze architectural approaches Step 6 is left unchanged, since it is based on results from step 5.

(7) Brainstorm and prioritize scenarios This step is separated into two parts. First, use cases/scenarios already covered by the prototype are evaluated in terms of user satisfaction and future (desired) use cases are voted for. Second, the results of that vote are used for clarifying existing specifications. The results and main (intermediary) results gathered and processed in this step remains the same, however the time-frame of its execution is altered again.

(8) Analyze architectural approaches Step 8 is left unchanged, as it basically says ”Repeat step 6, once step 7 is done.“

(9) Present results The findings of the (modified) ATAM execution are presented in this work and may be presented to industry partners if desired.

6.3 Results of the ATAM Evaluation

Using the modified ATAM (as described above), the eight steps were performed by parts of the CDL-Flex team and selected representatives from our industry partners to both gain a better understanding of the integration project’s goals and the effectiveness of the proposed architecture.

Present business drivers

From previous minutes of meetings and the long-term project agenda (ranging from early 2009 to the end of 2011), three drivers/issues towards a better solution were identified. All of them can be regarded as non-functional requirements regarding features of the existing solution approach: The tasks that need to be done can be done, but in an unsatisfactory way, meaning with high effort or a high chance of errors.

1. Failure to properly track changes and their propagation, causing misconceptions about a project's status (often by change requests from other parties, e.g. the customer side).
2. High costs to re-use assets from previous projects (i.e. mostly partial designs for components).
3. Strict coupling of the inter-disciplinary workflows to each software tool in use, restricting the flexibility of all groups.

Alongside, the main use case covered by the architecture is detailed in chapter 4 (see "Scenario"). It consists of three (for the illustration, only two are shown) disciplines exchanging parts of their designs ("signals") between each other, while trying to detect whether one group's changes affect the other groups' work. Thus, the main functional requirements to the integration solution are as follows.

1. Convert and transfer data instances from one of the tools into the respective other two models and vice versa.
2. Tracking the changes of one's work compared to the last known working copy and what changes are propagated to other disciplines.
3. Keeping older copies of the data available to to revert to those should current changes cause dissent between the engineering groups (in terms of breaking their work).

As indicated before, all of these functional requirements are/were covered without the solution from the CDL-Flex, but had severe shortcomings regarding the non-functional requirements.

Present architecture & Identify architectural approaches

The architecture/data model under review consists of five components:

1. Tool Data Models (TDM) which are pre-defined data models originating from the tools to integrate.
2. Tool Domain Data Models (TDDM) which abstract the TDMs to a higher level for better use "inside" the integration solution.
3. Common Data Models ("Engineering Objects") that are used to connect TDDMs with each other and allow for a generic, tool-independent view of the data instances. On a technical level, Engineering Objects are identical to TDDMs, with the only exception that

there is no underlying tool “in the outside world” that matches an Engineering Object’s data model.

4. Mappings are the “parsing configurations” to allow for an import and export of “raw” tool data into their respective domains and the only representation of the TDMs “inside” the integration solution.

5. Transformations are the translation instructions to convert data from one TDDM or Engineering Object into another.

The known benefits of this approach are that it allows for the relatively simple addition of both new tools and tool domains and the separation of the data model required for the use cases/scenarios from the tools’ data models. The known downsides of the design are its high(er) complexity and its narrow field of applicability.

From the previous steps there is a total of six identified goals the architecture has to serve. The non-functional one are referred to as: “Update Propagation & Notification”, “Ease of Refactorings” and “Reducing Tool Restrictions”. The functional requirements can be shortened as “Data Exchange”, “Traceability” (of changes) and “Versioning”.

- ”Update Propagation & Notification“ demands that specific data instances can be identified across model boundaries (usually finding the target of a change or request).

- ”Ease of Refactorings“ requires the ability to extract and batch-modify large sets of data.

- ”Reducing Tool Restrictions“ requires to check whether shortcomings of the tools’ data models are isolated and bypassed by the data model design or introduced into the integration solution.

- ”Data Exchange“ requires that (changing) data models can hold and exchange (tool) data instances.

- ”Traceability“ demands much the same as ”Update Propagation & Notification“, however mostly source-oriented.

- ”Versioning“ is not of immediate interest, but presents a general property all data storages in the integration solution need to have.

Regarding the design decisions made for this data model design, two main (modeling) concepts can be identified *Inheritance* (as TDMs can be regarding as subclasses of TDDMs) and *Abstraction* (as Common Data Models aim to aggregate ”interesting“ parts of the TDDMs). On the other hand, ”Global-Local-As-View“ can be identified as the dominant paradigm for the overall data modeling strategy, since the ”main“ models present TDDMS and Common Data Models treat each other as views of them, while being ”real“ data models used to access and modify the data instances.

Generate quality attribute utility tree

In the previous step, six aspects of the business goals were identified to be relevant when inspecting the data model design: “Update Propagation & Notification”, “Ease of Refactorings”, “Reducing Tool Restrictions”, “Data Exchange”, “Traceability” (of changes) and “Versioning”. Each of these goals harbors a set of expectations, in which way the integration solution of the CDL-Flex should enable one to achieve these goals and how to achieve them ”faster and cheaper“. However, not all of these goals have the same degree of immediacy attached to them, as “Ease of Refactorings” is something to be used in follow-up (engineering) projects after some data is already stored and handled with using the application integration solution. Also, the goals named “Update Propagation & Notification” and “Traceability”, although based on different use cases/scenarios require the same quality, namely tracking the links between data instances across tool (data model) boundaries.

This leaves us with four goals that are connected to several quality requirement attributes with the goal to better the user experience in mind.

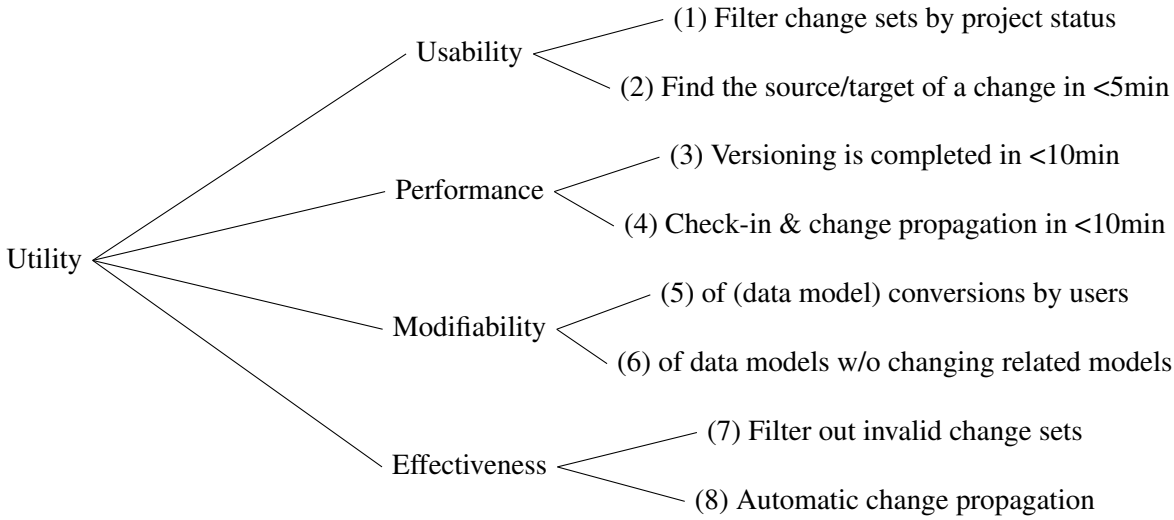
1. ”Traceability“ goes hand in hand with usability, as users may need to (re)view data changesets and effectiveness, as invalid data instances should be filtered out (or somehow made valid easily).

2. ”Data Exchange“ is connected to performance as well as effectiveness, since a functioning data exchange that does not slow down the engineers’ workflows is the core motivation to build an application integration solution.

3. ”Reducing Tool Restrictions“ is linked to modifiability and extensibility, as the data models ”inside“ the application integration solution have to embrace new features and should withstand a change of the underlying tool (or its data model).

4. ”Versioning“ also is linked to performance as it should not slow down the ”main“ user interactions.

Categorizing these attributes produces the following utility tree.



On further discussion, these eight quality attributes were specified in more detail as follows:

1. Being able to filter important/unimportant change sets by project status.
2. Be able to navigate/display the source/target for a local/propagated change set in <5min.
3. Versioning of change sets must be completed in <10min (should not stall later check-ins).
4. A check-in and the following propagation of data has to be finished in <10min (after user interactions).
5. Allow for modification of the conversions between data models by (power) users.
6. Allow for modification of data models without the need to change related models.
7. Automatically filter out invalid change sets before further propagation.
8. Automatically propagate changes across tool boundaries (after a check-in of data into one model).

From reviewing previous minutes of meeting and feedback from delivered prototypes, the assumed priorities (importance for the overall success) and risks (cost to satisfy) of the goals are as follows (sorted by priority, followed by effort):

#	Name	Priority	Effort
8	Change Propagation	high	high
5	Modify Conversions	high	medium
6	Modify Data Models	high	low
7	Filter Invalid Data	medium	high
1	Filter by Status	medium	low
2	Change Source/Target Lookup	low	medium
3	Fast Versioning	low	low
4	Fast Check-in & Follow-up	low	low

Using this overview, all goals of high priority will be further analyzed, regarding if or how they are addressed. (Also, all current medium and low-priority goals seem to be more about the UI and performance optimizations.)

Analyze architectural approaches

There is a total of three high-priority goals which have to be analyzed in further detail to answer the follow questions:

1. Does the proposed architecture satisfy the requirements imposed by the goal?
2. Are there any risks/conditions under which the requirement will not be satisfied?
3. Does satisfying the requirements impose any trade-offs on the architecture or the final solution which may hinder addressing other goals?

So, for the three high priority goals, the following quality attribute characterizations (QACs) were created:

Automatic Change Propagation across tool boundaries (after a check-in of data into one model).

1. Addressed: The data models are especially designed for the regular inter-exchange and conversions of data instances between all of them, the automatic propagation of changes however, has to be addressed by other parts of the (overall) architecture.
2. Risks:
   a Invalid input data causes indeterministic results or check-in interruptions.
   b A high number of tools and disciplines (>10) can lead to a nearly unmanageable data model design.
3. Trade-offs
   (a) A high initial configuration effort (data models and conversions).
   (b) The data model design needs to be explained in detail and made according to an extensive guideline to avoid inefficient or erroneous designs.

Modifying Data Model Conversions by (power) users.

1. Addressed: Both types of data model conversions are mere configurations, designed to be modified at any given time.
2. Risks:
   a Erroneous configuration of conversions lead to undesired data transferred into the target models.
   b Overly complex conversions slowing down the system.

c The number of necessary adoptions per project is underestimated, causing much ongoing configuration efforts.

3. Trade-offs:
   a Transferring data instance between major changes of data models and/or conversions is not guaranteed to work.
   b (Power) Users can easily break the system.

Modifying Data Models without having to update related models.

1. Addressed: The 3-tiered data model structure allows to modify parts of the design without having to update ”relayed“ data models in every case.
2. Risks:
   a Necessary updates of multiple data models are overlooked.
   b Local data models are updated often for no specific reason (”out of convenience“ or ”to look neater“), because it is cheap.
3. Trade-offs: The data model design is rather complex (3-tiered) and highly redundant.

This leaves us with the current conclusion that all major business goals (regarding data integration) are addressed by the data model design.

Brainstorm and prioritize scenarios

In the next step, all eight goals were presented to our industry partners. During the discussion about the importance of the goals, the industry partners caused two shifts of prioritization:

1. ”Modifying Data Models without having to update related models“ (6) was down-voted to low priority under the assumption that tools or their respective models do not change very often, especially not during projects. However, after showing counter-examples from the initial prototyping iterations, it was agreed that this goal should have a priority of medium at least (not high as it is not immediately needed for a working prototype regarding data exchange).
2. ”Being able to filter important/unimportant change sets by project status“ (1) was up-voted to high priority, as tracking and preventing changes to ”finalized“ data sets (i.e. ”Accepted by the customer“) is perceived to be a major cost-cutting factor (to which degree was unknown by CDL-Flex members until that discussion).

In addition, a new goal of medium priority was introduced with a discipline-specific scenario, called ”Creating and querying custom data hierarchies“. From the UI design point of

view, this refers to the user being able to categorize, sort and query data instances from their discipline by more then one ”hierarchies“. For example, an electrical engineer might sort their wired signals either by their physical location or by their ”control“ addresses which are not necessarily linked to the physical topology of a design. Such hierarchy access operates on existing fields of the data, but the user needs to create them on their own and should be allowed to switch between different ”perspectives“ when using them (e.g. using a tree structure which's nodes are the different parts of the physical or control address). Upon asking for a metric it was declared that creating such a hierarchy should not take longer then 15min, while switching to it and using it should delay the current task not more then 5min (since navigating through data usually is part of another use case). A quick draft of how such a feature could be implemented hinted that this is best be done using an isolated module in the UI (at least for the moment), as the central data model should not hold data used only for presentation purposes.

After said discussion, the new goal list looks as follows:

#	Name	Priority	Effort
8	Change Propagation	high	high
5	Modify Conversions	high	medium
1	Filter by Status	high	low
7	Filter Invalid Data	medium	high
9	Use Custom Hierarchies	medium	high
6	Modify Data Models	medium	low
2	Change Source/Target Lookup	low	medium
3	Fast Versioning	low	low
4	Fast Check-in & Follow-up	low	low

Analyze architectural approaches

Since the prioritization of business goals actually was changed, one more QAC has to be created (discarding the QAC created for the the now-medium priority goal is not recommended, as it might become useful in later iterations).

Filtering of change sets by relevance to the project status.

1. Addressed: The Tool Domain Data Models and/or the Common Data Models can hold status information fields independent from the tool data models.
2. Risks: An overuse of tool-independent fields makes using data instances in the tools uncomfortable, as meta-data is missing after data sets are re-imported into the tool(s).
3. Trade-offs: Complex data model.

Most notably, we see that the main trade-off of the proposed data model design style is a higher complexity in comparison to a 1-or 2-tiered approach, but at the benefit of serving the major business goals.

Present results

Summarizing the results from the previous steps, the validation of the data model design style using the ATAM yields the following results:

1. Six main business drivers were identified in the beginning of the ATAM execution, of which five are (partially) relevant when validating the data model design style. Out of these five goals, two are identical from the data modeling perspective.

2. These four business drivers were used to extrapolate a total of eight architecture goals, marking three as highly important for the success of the project.

3. The three quality attribute characterizations created from these goals supported the theory that the data model design style serves to satisfy the requirements.

4. ”Customer“ review caused a down-vote of one of the QACs, up-voted one goal to high priority and introduced a new medium-priority goal.

5. Repeated QAC creation again resulted in a positive evaluation of the proposed data model design style.

CHAPTER 7

Solution Concept & Validation Guideline Requirements

This chapter summarizes the requirements for guidelines to design, validate and update data model designs.

First, we provide for an overview of what the desired data model consists of and how these parts influence each other (in view of the chosen data model design). Using this structure, the challenges when modeling parts of the design are highlighted. Also, possible complexity considerations are given to ease later design of a modeling guideline.

Second, using a very similar structure, concerns regarding data integrity and validity are illustrated, mostly by recalling previously encountered modeling errors and design flaws. As before, this listing will serve as the basis for the following evaluation guideline, using past experiences and the (now better) structured data model to design a more formal inspection approach.

Third, follows a categorization of "in-life-cycle" changes to a data model that will be used to tailor the validation guideline from before to handle smaller changes as well as large-scale adoptions (without re-running the entire inspection).

7.1 Structuring the Design of a Data Model

Recalling previous chapters, the chosen data model style consists of five components: Tool Data Models (TDM), Tool Domain Data Models (TDDM), Common Data Models (CDMs), TDM-TDDM conversions (mapping) and TDDM-CDM/CDM-CDM conversions (transformation). Any structured approach to design a data model should

- propose an order which components to design when and
- how to design the respective components.

So, for a start, one needs to understand the varying degrees of liberty there are when designing the components and how they affect further design steps.

TDMs are perhaps the easiest candidate to start with: The data model of the local tools ("outside the bus") is pre-defined and can only be adjusted minimally in most cases, while on the other hand also determining much of the "upper" data models (TDDMs and CDMs) by being the source of (almost) all actual data. (Some meta-data may be added by the application integration solution, but the data payload is produced with these engineering tools).

TDDMs are next in the line with a little more liberty of design, as they are based on the TDMs, but are created from scratch to be a tool's gateway into (and out of) the application integration solution. Often, the first draft for a TDDM was an exact copy of the TDM only with some fields renamed for better readability. In addition, "normalizing" the data model by removing unnecessary or duplicate fields came next, simply to clean up a hard-to-read model. Likewise, fields that are known to contain multiple, semantically different information (often a simple "hack" by engineers to bypass shortcomings of a specific tool) were often split up to, again, make the TDDM "more readable". More often then not, in-between working on a TDDM, we saw it becoming virtually identical to another TDDM which in some cases convinced us (developers, researchers and industry partners) to merge such domains together (especially in cases of all domains offering the same functionality).

Although **CDMs** sometimes were present at the beginning of an integration project (out of the necessity that engineers shared data along disciplines already), most of these "grown" CDMs were of little use to our designs, if not a hindrance. Since there usually was a single "exchange data model" for all use cases, it was usually overloaded with fields from various TDMs, cluttering together data that was never used in combination with each other (a mistake we repeated in early phases of the CDL-Flex by using the VCDM). Later on, we found it a lot easier to create a separate CDM for each inter-disciplinary use case and merge identical (or highly similar) CDMs. This allowed for smaller and more robust CDMs, while the merging kept the number small enough not to make the overall design look (too) bloated.

Up until recently, the Mappings and Transformations were done at once, after all the data models were designed. The main motivation was that doing this was tedious anyways and doing it all at once was easier to get over with for most developers. Unfortunately, the negative side-effect of this is that any design failures in the TDMs, TDDMs and CDMs were only discovered when specifying and testing the data conversions (the same goes for imprecise conversion rules, as people tend to forget what computers can do automatically and what requires "real" intelligence). The **Transformations** deal with TDDM/CDM-CDM conversions, any shortcomings here usually origin from mistakes of the application integration developer (or a direct colleague). So, doing this part at the end of the design phase was a minor issue (as the "guilty" person can be expected to be easily reached by the developer). Issues with the **Mappings** however, only stem from shortcomings in the initial specifications and data model of the industry partner, so designing them as soon as possible is more desirable (if only to reduce any effort wasted on an infeasible TDDM or CDM).

7.2 How to Validate a Data Model Instance Upfront?

Recalling the aim of this work, there are three major objectives: Creating a data modeling style for large integration projects, validating a design (using the same style) and validating future

changes against such a design. Since prototyping and an inspection using the ATAM strongly indicates that the chosen data model design style from chapter 6 suits the purpose, it remains to find a way to check whether a specific execution of the design style was done correctly. This (sub)section deals with the various aspect of the data model and the errors they may contain, summarizing the needs we found for such data model validation (from mistakes and errors in previous projects). The overview is later used as the basis for a validation guideline. The reasoning being is that a guideline that covers all specified aspects and (known) errors is an effective solution for validation tasks.

Below follows the listing of data model contents and the complexity of aspects of the data model to allow for a structured approach when drafting validation guidelines later on.

Concerning validation tasks, each of the data models can contain up to four different types of fields:

1. Key fields - used for identification of data instances (either in a specific tool or across tool boundaries)

2. Use Case fields - parts of a data model that are specifically queried or manipulated during an inter-disciplinary workflow, not by using any of the integrated tools, but via the integration solution

3. ”Related“ fields - data that is stored and manipulated in more then one tool over the course of a project, but is neither (part of) an identifier nor used in any of the integration use cases

4. Other fields - all remaining data from a tool which is not an identifier, is not present in more then one tool and is not used in the integration solution (but may be needed for a valid im-and export of data to/from a tool)

Using this basic categorization of the data and the underlying design style, we propose an inspection procedure that deals with the different layers of the data model ordered by complexity (ascending), leading to the following validation steps:

1. Key Validation - to ensure that each (sub)model at least allows for a sound identification of its instances and that references between data (sub)models are feasible.

2. Field Validation - next to check whether the content of each data (sub)model ”makes sense“, i.e. that each data field has a known use.

3. Mapping Validation - aims to guarantee that data can actually be transferred between different data models.

4. Transformation Validation - is motivated by the same reasoning as Mapping Validation, but between different (engineering) disciplines (which is the main purpose of the entire solution).

5. Goal Validation - as the top level check is targeted at the usefulness of a given data model design with respect to a set of defined use cases, i.e. whether the known (functional) requirements are satisfied by the design.

7.2.1 Key Validation

The first series of checks revolves around the subset of the data which is the smallest most of the times: The primary identifiers. During prototype iterations, the following issues were discovered regarding the identifiers:

- ”Lost“ keys - identifier fields were not ”transported“ from the tool/domain model to another domain/common model
- Wrong keys or key mappings - the wrong data fields were treated as keys or the ”transported“ into the wrong fields of a target model
- Invalid conversions - identifiers are represented differently across different tool, domain and common data models which sometimes was not specified clearly
- Unavailable keys - if a tool does not export its identifiers, own ”internal“ identifiers have to be created and for every data im-and export (and somehow kept track off)

Surprisingly, such a variety of errors can be checked for with a very simple procedure. This comes from the fact that any errors regarding identifier fields can be checked for without looking at any other fields. The main issue here comes from the difficulty of keeping a clean track of one’s own progress when digging through all data models, mappings and transformations. Any solution to this challenge has to provide for a clear sequence to check for the identifier fields and could easily be used as the base of following inspection runs. (See the next chapter for a list of these steps.)

7.2.2 Field Validation

Once the ”traceability“ of data instances between the models is validated, ”all the rest“ of the data models has to be inspected, verifying the ”justification“ of all other fields. All in all, the correctness of semantics for each field may not be verified by this approach, as this requires expert knowledge of the used tools and their intersecting use cases. Such an inspection only serves the purpose of validating a sound model based on the use case and tool model specifications given to one or more integration application developers. To this purpose, tool, domain and common data models may be inspected in a different way.

Tool Models do not need any further inspection here, as they are only present in the mappings, which can be checked later on.

Domain Models should be the main focus for the next validation phase, as they represent the integration developer’s understanding of the underlying tool and are the basis for the common models. In addition, they serve as the gate keepers for most ”actual“ data in the final application integration solution, being the first piece of software ”inside“ the solution to hold the data. During our past (and running) integration projects the most common errors for domain data model design made were

- irrelevant fields being transferred into the domain (i.e. fields not needed for a re-import, for exchange with any other tool and any use case).
- tools being ”pushed“ into a separate domain while only acting as an additional data source for other tools
- duplicate fields (placed for formatting or exporting issues) transferred into domain (often causing inconsistencies of the data itself or inconsistent reference to what field is the ”original“ one)

Common Data Models can be looked at after the domain data models have been verified (and probably been merged again) to only contain “useful” fields, i.e. fields that either originate from an actual tool or are part of a use case. So, under this assumption, a reduced version of the domain data model check could be applied. (See the next chapter ”Solution“ for a list of these steps.)

7.2.3 Mapping Validation

Since a previous checking of the identifier fields already enforces testing the mappings, this may seem redundant, however due to frequent oversights when importing and exporting tool data in past projects, checking the mappings should be done a second time, however from a different perspective, for various reasons.

- Although the ”computability“ of the mappings and the ”traceability“ and have been checked already, both industry partners and developers often failed to agree what “clear specifications” (of a mapping or transformations) means and/or missed to consider transferring all identifier keys from one model to another (especially if a model holds multiple of such key set from different sources).
- Use case relevant fields somehow tend to be “forgotten” in mappings (and transformations), perhaps due to the fact that they were often adapted for other use case or the most problematic fields to transfer, sometimes being delayed “to tackle them later”.
- If a data model had to be extended “inside” the integration solution (e.g. for adding meta information), persisting such fields in the desired target domains or transferring them to the common data model was often overlooked (maybe because except for one use case no-one required those fields).

7.2.4 Transformation Validation

Transformations are data model conversions meant to support inter-disciplinary workflows, therefore the validation of transformations should ensure consistency of data instances and traceability of data flows. The consistency of the data flows depends (mostly) on the correctness of the initial specifications, which is outside the scope of these checks. The only exception is the case that such erroneous specifications contradict a use case, i.e. that data is accessed and modified across tools boundaries that was not made available via the common data model, which is dealt

with in the next inspection phase. However, due to previous checks, the transformations should have already been tested for traceability, following checks should “test” the transformations themselves, i.e. if transformations follow the given specifications.

7.2.5 Goal Validation

The final inspection phase should check the feasibility of the designed inter-disciplinary use cases with two (relatively) simple steps. This requires a (bare) use case specification to be present at least (but designing a model beforehand would not make much sense anyways). Such checks have to cover the fact that each use case should result in at least two domains being involved (requiring a transformation) and that one domain (or the common data model) is used to provide access to new data or stores the same data (originating from a source from another discipline).

7.3 Handling Updates to the Data Model Instance

Designing and validating initial data model designs alone allows for a successful creation and deployment of an integration solution that will work only for a limited amount of time. Changes in the workflows and tool models or adoptions of the solution to other teams potentially demand for a modification of the original design. With such changes, there is a chance that the data model design will become broken, threatening the validity of the integration solution. The easiest way to ensure that this does not happen would be to repeat all inspections as if creating the data model anew. However, if this is done for every change request, there is a high risk that the checks will simply be ignored after a few iterations, especially if they yielded “nothing bad has happened”-results in the previous runs. Therefore, a reduced set of checks has to be made available for the different types of change requests. (See the next chapter for a detailed list of these steps.)

Below is a classification of changes to the data model that allows to check the “scope” of the change. This classification should later help to reduce inspection effort for each change type (based on the implications and side-effects of them).

7.3.1 Single-field changes

These refer to changes in any data model, as long as the change does not alter more then one or two fields at all. Usually, this indicates simple renamings, additions or other minor fixes to an otherwise stable data model. Doing pin-point inspections for this type of change may cause some initial search effort, as listing affected mappings/transformations and workflows may be hard to come by, but it is still easier to identify these subsets of the design then to re-run the entire validation.

7.3.2 Tool-wide changes

Such a change type describes one that alters a tool’s data model, affecting at least one domain and a mapping as well, but likely causing minor adoptions over the entire data model. Change request to a tool’s data model, the removal of a tool or the addition of another tool (to an existing

domain in this case) into the integration solution are major milestones during the development, as they test how "farsighted" the data model design was first-hand or how easily it can be adapted. Such modifications occur less frequently during a project's life-cycle and are usually known it advance, as part of two possible scenarios:

- exchanging one tool for another (e.g. due to terminated support of the old one, new features expected of the new one or cost-cutting) or
- version updates of existing software with wide-range changes to the data model, features and user experience changes.

Identifying the impact of such a modification can range from a short check to a full re-check of the data model, depending on the dependencies between the different domains.

7.3.3 Domain-wide changes

Any change called tool-wide is a major re-design of the integration solution and will most often cause an entire re-evaluation, but under special circumstances, a reduced set of checks can be applicable. Either modifications of domains are motivated by transitive updates from tool model changes or from specifications changes (i.e. new features expected from a domain). In the first case, simply starting from the tool data model changes and adding checks regarding the transformations/mappings should suffice, while in the second case a full re-run of the original inspection is most likely.

7.3.4 Changes to the common data model

These are quite like changes to the tool domains, may either be based on updates of "underlying" models or be based on extensions to the common design, aimed to enrich the integration solution's capabilities. As with tool domains, the first case should be covered by preceding runs of inspections, leaving the case of "top-down" changes to cover for. In this case, a reduced set of checks can be applied to effectively validate that changes to the common data model are propagated into "lower" models, so that these changes actually affect the behavior of the integration solution.

CHAPTER 8

Results

This chapters summarizes the findings of this thesis' work to present the proposed solution. Based on the results from the ATAM evaluation of the architectural flavors in chapters 6 and the design structure, change and error types from chapter 7, this chapters provides the suggested data integration style. Further, it lists the recommended steps to derive a data model from the initial requirements and sample data. Also, the guideline drafted before is given in full detail to allow for a validation of a designed data model instance without the need to fully implement the integration solution beforehand. In addition, the extension of the same guideline to deal with changes of the data model is explained. Finally, the suggested solution is compared to the initial challenges in order to validate that all of them have been addressed and solved.

8.1 Final Modeling Style

After comparing a set of candidates, one data model design style was selected (more precise: discarding the others for ineffectiveness) as a candidate for a solution. Below (figure 8.1) is a detailed description of the area of application for said style as well as step-by-step instructions on how to use it. (Later subsections deal with the validation and updating of a specific data model instance.)

8.1.1 Prerequisites

The proposed modeling style is meant be in environments where engineers (or teams thereof) from different disciplines have to collaborate over a longer period of time to create and design complex engineering artifacts (e.g. power plants or factories). If the working environment matches such a scenario the following initial data should be gathered:

- a list of used engineering tools
- (optional) a short (informal) description of who does what with these tools

- data model descriptions/specifications for these tools
- the use cases/scenarios/workflows that require inter-discipline collaboration
- (optional) the issues related to these collaborations

Note that the "description of who does what" is not strictly necessary, but usually allows for a better "tool domain" design later on and also helps identifying if a tool is used for more than one purpose, sometimes indicating multiple disciplines and domains being worked on in a single tool. The most prominent example for such a case might be Excel, which is often used as a multi-purpose environment especially with macros, forms and templates.

With this data, one is enabled to start the initial design steps and should have a basic understanding of the working environment.

8.1.2 Designing the Model

As explained in the previous chapters as "Multiple Domains and one Engineering Object", a 3-tiered data model design is created in a sequence of steps. Below is a short list of these steps, followed by a more detailed explanation.

1. For each tool, create (or simply read if already provided) a description of each tool's data model(s).
2. For each tool data model, plan a corresponding domain with its own data model, capturing the tool data model's purpose.
3. (Optional) If some tools happen to have the same purpose and similar data models, consider merging their domains.
4. Create mapping descriptions between these tool data models and their matching domain data models.
5. For each (inter-disciplinary) use case, create a common data model that holds the data which is manipulated and/or queried during this use case.
6. (Optional) As with the domains, if there are common data models with a high likeness to each other, considering merging them.
7. Create transformation descriptions between common data models and domain data models that are involved in a use case.

Tool Data Model Design This step simply serves the purpose of documenting existing data models or getting hold of their specification for every tool. Ideally, one has access to both sample data, which will also be used for later tests, and a formal description of the data model. In most cases, the tool's data model is static, but sometimes can be modified, at least from the view of the integration solution, e.g. by changing an export filter. However, should it be possible to actively change a tool's data model, this will ease later tasks enormously as such a tool can be

enhanced with additional key fields (for cross-tool matching) or even hold “meta-fields”, such as version numbers or status tracking flags. Note that a single tool may hold several (unrelated) data models if it is used for different tasks. No template is provided for this description, as our industry partners preferred (and usually enforced) vastly different documentation styles, ranging from simple UML [37] ER or class diagrams to extensive full-text descriptions. However, it is mandatory to include at least the following information

- the names and types of all fields
- the “usage” of the field, i.e. whether it is an identifier, a “normal“ field or auto-generated
- a short description of what data the field holds, i.e. its ”meaning“

Tool Domain Data Model Design For the initial design, each tool data model is considered the sole representative of its tool domain. The rationale behind this approach is that usually (for one specific team in a specific project) only one kind of tool is used for any task and the option to exchange that tool is not an immediate concern. (However, adding a domain-unique and tool-independent identifier has proven useful in many cases.) Under most circumstances, the domain’s data model will look much like the tool’s data model, simply “prettifying” the data model, by

- making field names more self-explanatory,
- splitting up fields containing multiple information,
- merging information split along multiple fields,
- removing duplicate fields,
- eliminating null-able fields and
- omitting auto-generated fields that do not contain required information.

Also try to name the domain either after its main purpose or after the dominating data instance, so that the similarities (or differences) to other tools are explicitly stated at least in one place. As with the tool data models, no specific template is provided for this description (for reasons, see above), the same goes for the minimal information to include.

Merging Similar Tool Domain Models This step will appear quite obvious if necessary, if more than one tool is used to achieve the same goal. There are three scenarios for such a dual/multi-use of tools:

1. The tools are used on virtually identical data models (and their instances), but complement each other in terms of efficiency for certain operations or
2. the tools create and/or modify different subsets of a data model, but cannot be efficiently broken down into smaller data sets, e.g. because they share too much common data or

3. the tools operate on different subsets of the data instances, bearing the same model, but never interfering with each other.

All three of these scenarios stem from short-comings of the respective tools which the engineers who use them would like to overcome, as their view on the "big picture" is hindered. While the application integration solution cannot overcome these issues in the tools, it can at least help to bypass them elsewhere.

Mapping Specification Although this should be a simple by-product of the former steps, it is noteworthy to explicitly document how data from a tool is converted into tool domain data and vice versa, if only to ease later inspections and the developer's work. Such a description must contain a list of

- all fields from the source model (tool or domain)
- all target fields (tool or domain)
- a deterministic and computable description of how to convert one or more source fields into one or more target fields (it is highly recommended to restrict the conversions to either 1:n or n:1 consistently)
- a full sample for each single step
- a valid test data sample for tool and domain data (valid here means that the data will be accepted by the tool or domain "as it is", i.e. without further modification)

Below, there is a sample template for such a description (surprisingly, even within their own specific documentation preferences, developers and industry partners often failed to produce valid mapping descriptions, especially if the tool's data model was hard to understand for non-experts).

Common Data Model Design All use cases that involve more than one tool domain have a subset of their respective data models via which they interact with each other, sometimes only a common identifier and a single data field, in other cases the entire data model. In this step, the subset of data has to be identified and modelled the same way as a tool domain (see above for further details).

Merging Common Data Models As with the tool domain data model design, it may happen that multiple common data models look similar to each other or to a specific tool domain. If this is not the case and most use cases demand for very distinct common data models, this can indicate issues with the current level of collaboration (or simply very strict policies regarding data access). It is quite possible that a common data model that has just been created will be removed again and its task of mediating a use case gets taken over by a tool domain. On the other hand, it may also occur that the merger of two common data models obsoletes the role of a tool domain for a specific use case (especially if the data exchange is not time-critical).

Transformerdefinition Version V.1.0.0

[redacted]-OUT-1.doc

Supported Tool: EPlan

Transformer Name: [redacted]-OUT

Reference Test File: [redacted]-OUT.txt

Industry Partner

2011-08-19 Contact: [redacted]

2011-08-19 CDL: Florian Waltersdorfer, Richard Mordinyi, Dietmar Winkler

Table Description

Anzahl der Kopfzeilen: 1 (siehe Fieldnames)

Trennzeichen zw. Feldern: „;“ (Semicolumn)

Test Data Set				Transformer / EDB	
Field name	**Notes**	**Format**	**Example**	**Relevant**	**VCDM Assignment**
PLC address	Field separator between Addr-* is “.”	String-###.##.##.#.##	OUT: “010.01.02.0.00”	yes	
Addr-1			IN: “010”	yes	componentNumber
Addr-2			IN: “01”	yes	cpuNumber
Addr-3			IN: “02”	yes	peripheralBoardAddress
Addr-4			IN: “0”	yes	InputOutputModule
Addr-5			IN: “00”	yes	channelName
Data type	Relevant column, but empty			yes	
Symbolic address	Relevant column, but empty			Yes	
Function text		String	IN/OUT “U# - Temp. Stator Winding / phase U / centre”	yes	FunctionTextOne
Signal number	Field separator between Sig-* is	String-	OUT: “0#.MKA20.CT001.XQ01”	yes	

Figure 8.1: Example of a mapping specification.

Transformation Specification In the last step, the documentation about how to convert data from one domain into another domain’s model (one way or both ways) is created. Like the mappings, this description has to contain

- a source model
- a target model
- a deterministic and computable description of how to convert one or more source fields into one or more target fields (it is highly recommended to restrict the conversions to either 1:n or n:1 consistently)
- a full sample for each single step
- a valid test data sample for target and source (valid here means that the data will be accepted by the domain “as it is“, i.e. without further modification)

Unlike the mappings however, a single template has not proven successful yet, as the conversions between domains and common data models can become much more complex, requiring an extensive description and references for some of the fields that do not fit well into a tabular layout. (From our experience, simple full-text or pseudo-code descriptions work best, but unfortunately defy attempts for a standardized format).

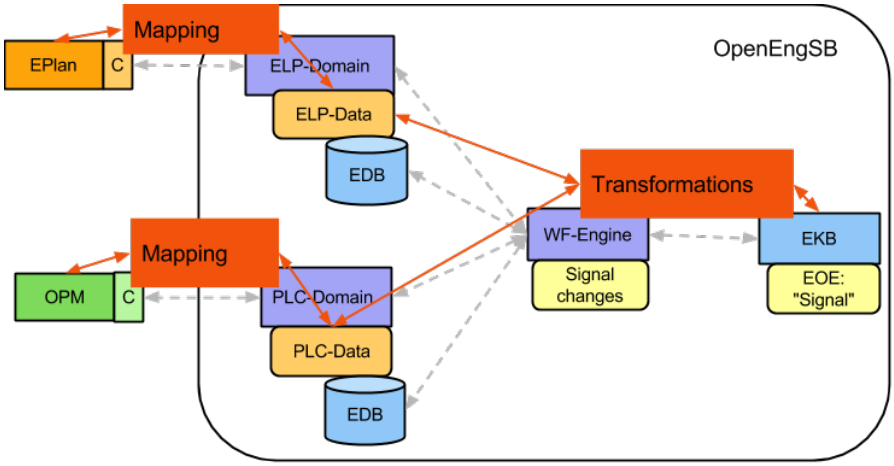


Figure 8.2: Data conversion separated into tool-domain and domain-domain matters

Result The resulting design/specifications derived now consist of the Tool Data Models (TDM), Tool Domain Data Models (TDDM), Common Data Models (CDM or ”Engineering Objects“), the TDM-TDDM Mappings and the TDDM/CDM-CDM Transformations.

8.2 Design Validation Process

Validating such a 3-tier data model design is a time-consuming and repetitive task with little chance of producing reliable results if not done in a structured way (see previous chapter). Therefore, a 5-phase inspection sequence was created to ease this process: Each phase relies on information from former phases to reduce the amount of re-occurring checks and to gain a better understanding of the overall design (and its shortcomings, if any). Should, during such an inspection, any phase uncover errors or contradictions, the reviewer has two options:

- Note the issue and introduce ”fixes“ to the design by stating own assumptions and continue the inspection.
- In case of too many or too severe design defects, abort the inspection and present the designers/developers with a list of found defects.

This allows to finish the inspection in the case of all ”minor“ flaws without having to wait for additional feedback from other parties or to stop the inspection soon if the overall design is flawed.

The said validation phases are as follows:

1. Keys
2. Fields
3. Mappings

4. Transformations
5. Goals

Key Validation The initial phase aims to check if all models hold valid primary keys and to ensure the traceability of data instances between the models. For all data models (tool, domain and common)

1. Check if every data model contains at least one identifier ”for itself“.
2. All data models which are ”connected“ via transformations and mappings, starting from the tool models, one of the following must hold:
   - The models must either contain the identifier from the other model or
   - The transformation/mapping between them must compute the respective identifier of the other model
3. For the last step, use identical sample data from all tools to test the given transformations and mappings; all resulting data should contain key fields and equal the given sample data (inspect only identifier fields in this step).

Field Validation The second phase checks the ”justification“ for all data present in the models, eliminating unused or duplicate fields and unnecessary domains.

For all tool domain data models:

1. For all non-key fields, identify their ”origin”, i.e. why they were placed in the domain model. Every field must either be
   - based on a tool model field (not necessarily from the tool connected to the currently inspected domain) or
   - used during a use case (and therefore should reside in the common model as well).

   All “origin-less” fields should be considered for removal.
2. For every tool domain, verify that a valid export of this tool’s data can be performed, i.e. the tool can import the produced data.
3. For every field per tool domain, check that there are no duplicates inside a domain model.

Regarding the duplicate checks: As there is no way of automatically detecting such duplicates, this is best done by checking a small set of test data together with requesting a short textual description of each field’s role.

The Common Data Models are only looked at after the domain data models, checking that each field in a given common model

1. originates from a domain or a use case and
2. has no duplicate in the same common model.

Mapping Validation The third phase is mostly a re-run of critical checks from the previous phases and includes the fore-work for the last phase (tracking use case relevant data).

For all mappings,

1. (manually) simulate them with given test data to show that they are sufficiently specified to be performed by a computer,

2. check that each of the underlying tool's identifier fields is transferred to the domain model,

3. verify that all use case relevant fields are either

   - present in a mapping, i.e. have a "valid" source or
   - are explicitly modified/added by the integration solution (originating from a common data model) and

4. (optional) see whether the mappings can be "inverted", i.e. that the source data can be reproduced from a mapping result.

The last step is optional (for some models) because any tool may act as a pure data source, but this should be known in advance (either by design or from previous inspection phases), otherwise the use case specification preceding the data model design is most likely flawed.

Transformation Validation The fourth phase is entirely optional, consisting only of a test run of the transformations (this should not uncover anything new, but can serve as a way to detect sloppy "implementation" of the design).

1. (Optional) For all transformations, (manually) simulate a run of the transformation from one domain to another (test data should be available from the tests above) and compare the results against specified test data instances.

This test is considered optional, as its main focus is not to double-check the specifications, however it may help to uncover severe specification errors.

Goal Validation The last phase guarantees that the design in question can serve the use cases from the initial specification, relying on the results of the former phases.

For every use case, verify

1. that at its end, one or many fields from a specific domain (or common data model) is/are accessed and/or modified and

2. that at least one transformation is executed somewhere in the use case.

8.3 Change Validation Process

Validating changes to the three types of data models and their mappings/transformations is extremely repetitive considering the scope of a typical update (small) compard the effort to re-run the entire inspection. As already hinted in the previous chapter, a subset of the checks suffices in many cases to determine whether a change impacts other parts of the design not considered before or is entirely without effect. Depending on the scope of the change, variants of the inspection are needed to be run again, categorized as follows:

- Single-field changes
- Tool-wide changes
- Domain-wide changes
- Changes to the common data model

Regarding the procedure itself, the same basic rules as for the ”normal“ inspection apply: The reviewer can either make own assumptions to continue the run in case of ”small“ defects detected or abort in case of severe defects found. However, unlike the initial run, any assumption placed to ”fix“ minor flaws has to be treated as an update, therefore enqueueing a re-evaluation in the respective scope (see above). This is to prevent that a re-evaluation itself introduces regression bugs into the design or succeeds on a broken design ”with just one small fix that cannot break anything“ (but instead has a series of follow-up bugs not seen, because the required inspection steps were skipped).

Single-field change requests for a single field are most common, as they are often the result of specification corrections, iterative expansions of the integration solution and simple bug-fixes due to previous oversights in the design. Using the following three steps, one can validate the new data model design.

1. Depending on the type of the field, fully repeat either
   - the “Key Validation” phase or
   - the “Field Validation“ phase.
2. For all mappings and/or transformations the field is used in (either as source or target), repeat the respective mapping/transformation checks.
3. Next, identify whether or not the field is part of any cross-disciplinary workflow, if so repeat the “Goal Validation” phase.

There are two possible results for these checks: Showing only renamings of already set fields or failures during the checks. In the first case, the proposed changes are likely irrelevant and should be either discarded or the motivation for their proposal (i.e. naming conventions) should be discussed. In the second case, every mismatch or failed check indicates that some updates have not been properly distributed in the data model design and that further updates are necessary (usually in the “target part” of a failed check).

Tool-wide changes are somewhat harder to check, as their impact on the overall data model design can be quite severe, with a high chance of some of the transitive modifications in "related" data models being missed. So, for every modified or new tool, proceed as follows:

1. Check if the tool data model is present (updated or new).
2. Perform the "Key Validation" phase for the changed tool model, its domain model(s) and all related common data models.
3. Perform step 1 of the "Field Validation" phase for all domain models.
4. Perform steps 2 and 3 of the "Field Validation" phase for all domain models originating from the current tool model (should not be more then one in most cases).
5. Perform the "Field Validation phase" for all common data models directly linked to the domain model(s) of the current tool model.
6. Perform step 1 of the "Mapping Validation" phase for the current tool model.
7. Repeat the "Goal Validation" phase.

There are two possible outcomes from these checks; indicating only renamings of existing fields or mismatches during the checks. In the first case, the tool model changes remained "local", i.e. did not affect the overall data model (which is the most desired scenario). In the second case, every mismatch or failed check indicates that some mapping/transformation or target models will need additional updates (those that cause failed checks during this re-run) and need updating.

In case of the removal of a tool, another procedure is recommended.

1. Check if all domains still have an underlying tool, if not, remove them immediately (with all "related" mappings and transformations).
2. Repeat the "Key Validation" and "Field Validation" phase.
3. (optional) Repeat the "Goal Validation" phase.

In theory, the last step should not be necessary, as a tool whose data is still needed should not be a candidate for removal, but depending on the number of inter-domain use cases, it can be very advisable to cross-check against such an oversight.

Domain-wide changes Modifications of domains are either motivated by transitive updates from tool model changes or extensions of the specifications (i.e. new features expected from the domains). In the first case, the checks related to the original cause of the updates should be finished and any domain data model modifications triggered from these checks should be handled in one of the follwing ways

- Perform the "Transformation Validation" phase for the affected domain models (non-optional in this case) and

- Perform the "Mapping Validation" phase for the affected domain models.

Note that the order of the checks does not matter here and that any field validations should have already been done before. However, in the second case or if the updates to any domain exceed two or three fields, this indicates a major re-design of the integration solution happening, in which case it is recommended to repeat the entire validation process as if a completely new data model design was to be tested. This is motivated by the fact that the domains hold the core data model of the integration solution and a re-evaluation of the entire design is likely to happen anyways, only with the checks performed in a less organized way otherwise.

Changes to the common data model that did not originate from domain model changes (which already caused a full re-run), can be checked using a very narrow top-down approach to ensure they fit into the overall data model design. For all updated common data models,

1. perform the "Transformation Validation" phase for all domain models linked to the current common data model (non-optional in this case).
2. perform step 1 of the "Field Validation" phase "inverted", i.e. do not verify an origin for each field, but for all modified common data model fields now check that they are either
   - transferred to a "target" tool data model or
   - to another common data model or
   - are queried during a use case (this means if neither of the first two checks succeeds, marks such fields for step 3).
3. repeat the "Goal Validation" phase.

Should a common data model be removed, the same procedure as with a domain removal has to be performed (see above).

8.4 Evaluating the Guidelines

Having established said modeling approach and validation guidelines, we again look if they offer a good solution for the challenges stated in section 3 by checking for the requirements from section 7 for each of the three procedures. Therefore, what follows is a reflection whether and how those requirements were served. Each of the three research issues is reflected upon in a separated section, with the same structure. First, a (short) description of what was done in the work to find a answer to the question is given. Second, the results of said work are summarized to allow for a decision whether or not the question could be answered in a satisfying manner.

8.4.1 Modeling Approach

Based on the experience gathered from previous prototypes [76], it was known that a new data modeling style was needed. As the initial requirements had been extended by this point, literature research regarding known data integration styles, strategies and technologies was performed. This brought up the knowledge that there are no commonly accepted and wide-spread

"best practices" or patterns for data integration, only recommendations how specific technologies can be used to build middle-ware systems. Based on these insights, two variants for application integration were tested via prototyping to find out if any of those could help to derive a data modeling style: A service-based style, as popular with Enterprise Service Bus (ESB) designs, and a domain-based approach, as favored in the OpenEngSB project (loosely related to the previous VCDM solution). Both prototypes did not yield a sufficiently flexible and robust solution, but gave enough insights to derive a 3-layered data model design which seemed to be a viable candidate for a good solution. In addition to the review of the initial requirements of the data model, an ATAM-based inspection of the proposed data modeling style was executed, which also confirmed the viability of the proposed data model design as a good solution. So, for this data model design style to be a good solution, the following criteria have to hold:

1. It must allow to hold data model representations of proprietary software.
2. It has to allow for the exchange of (multiple) subsets of data instances across tool data models (see the scenario "Signal Engineering Review Circle" under "Method").
3. Tools need to be added and modified without causing a compete overhaul of the data model (removal of tools without replacing them with a software used for the same task is not a concern here).

The 3-layered data model design proposed for a solution serves these requirements as follows:

1. The Tool data models (TDM) are not represented directly into the application integration solution, but are present in the mapping configurations used to convert them into valid data for the respective Tool Domain Data Models (TDDM) which stores and versions data instances of all tools inside the system.
2. The Common Data Models ("Engineering Objects") are data models designed especially for the exchange of (sub)sets of discipline-specific data across tool boundaries.
3. Adding or modifying a single tool domain causes only limited changes "outside" of said domain, which is presumably the best degree of change minimizations that can be achieved.

8.4.2 Validating a Given Design

After the initial review of the data model design style (with the ATAM), a guideline was specified, allowing to create a specific data model design from scratch. The same guideline was used to identify common errors when designing a data model, in addition to experiences from earlier prototypes and known change requests and regression errors when updating data model designs (which were deployed for industry partners). As the validation process should be performed early on (in the pre-development phase) several known inspection techniques from software quality assurance were reviewed, with some of them being adapted to fit into a proposal for a data design validation guideline. So, for the validation guideline to be thorough and efficient, a set of criteria has to be fulfilled:

1. All components of the data model design have to be inspected at least once.

2. All known errors (see “Validating the Data Model Design” under “Method”) need to be covered by the tests.

3. Any errors detected when inspecting a component should trigger inspections in “related” components.

4. Redundant checks should be avoided.

The proposed data validation guideline serves these requirements well in the following ways:

1. The inspection itself is split into several phases, with each split again into sub-phases for each component type. Also, as the reviewer is guided through the layers of the data model (and therefore also most likely follows a “typical” data-flow), the overall understanding of the data model design increases during an inspection.

2. Each single step in the respective phases (per component) was made having a single type of error in mind. This allows for short and simple checks that (un)cover one error at a time, The exception to this approach are the inspection phases for mappings and transformations, as any errors in their specification (that did not originate from previous errors) are most easily discovered by simple “running” the conversions (whether manually or with tool support is of little concern here, despite of what the guideline says).

3. The propagation of data model design flaws was dealt with differently than initially expected. The guideline (unlike a typical review) demands the reviewer to “fix” any small defects during the inspection and report them afterwards. This allows the reviewer to finish an inspection, even if small errors are uncovered.

4. The entire review process is structured in a way that many checks which occurred during earlier phases ease the work in later phases by creating “safe” assumptions about the overall data model design. Especially the “Goal Validation” phase is relatively short, as the “sanity” of the data model has been checked for most parts already. This means that the only remaining inspection goal in the last phase is to ensure the “correct” use of the data model (to execute the actual use cases).

8.4.3 Handling Changes to the Design

The validation procedure proposed before aims to guide through a full inspection of the designed data model with as few redundancies as possible. This works well when reviewing an entirely new data model, but is an over-the-top approach if one only has to review an existing model after a change request. Therefore, an extension to the same guideline was made which should cover such update scenarios. In the first step, the possible change requests where categorized, depending on their impact on the data model. In the second step, parts of the initial inspection guideline were selected for repetiton. Those steps of the validation procedure aim to cover all consequences introduced by changes to the data model, but should not (always) cause an

entire re-run of the validation guideline. In the last step, an additional "final check condition" was added to the change evaluation process to ensure that the results of said checks are either identified as "local enough", so that no further checks are needed or that they trigger additional checks of other components. Reviewing on this procedure, it now needs to be checked whether:

1. This procedure allows to distinguish between correct executions of change request, i.e. valid changes to the data model, and changes that break the design.

2. Changes that do not alter the data model in any significant way are handled differently then changes that change the behavior and structure of the data model.

3. It requires less effort to validate a data model update than a complete re-run of the data model validation procedure.

From the very design of this process, it becomes quite clear that these requirements are met:

1. Any update causes a full review of the parts of the data model that changed, excluding only non-related data models and conversions. So, if such an update leaves "gaps" in the data model, i.e. changes that were not executed "all the way through", or renders a use cause un-doable, it will be uncovered. This is guaranteed because the entire data flow of the updated model is checked and as well as its role in the use cases.

2. As with identifying poorly executed changes, a similar procedure is triggered for irrelevant changes, as those do not escalate the checks to the next component. The previously mentioned "final check condition" of each change evaluation sequence also exposes all changes that only affect a single component of the data model, which indicates that the change is purely cosmetic and can be discarded (except for when the "customer" demands the renaming of field names for other reasons).

3. By design of the guideline, in most cases, only parts of the full data model evaluation procedure are executed, which reduces the effort compared to a full re-run.

CHAPTER 9

Discussion and Limitations

This chapter reflects on the results from chapters 7 and 8 to argue that the proposed solution is a feasible (and satisfiable) outcome of this work. The second part of this chapter highlights unsolved challenges or new ones which have arisen during the implementation and validation of the prototype and guidelines to explain the limitations of the solution.

9.1 Discussion

Before summarizing the results of this work, a short re-visit of this work's initial motivation and resulting challenges is given. Following is the summary of the evaluation results to show how the proposed solution suffices this work's goals.

9.1.1 Modeling Data Integration Solutions

The core scenario spawns from inter-disciplinary engineering workflows in large projects. Members of different engineering departments, using different tools and having similar (but not equal) data models, want to exchange, review and modify their respective work-sets without having to abandon their current tools. This requires an application integration solution that connects the used tools with each other, allowing for a (possibly) seamless exchange of data between those tools, using on-the-fly data conversions. Furthermore, it should provide for some kind of merge mechanism that enables the engineers to actively receive the changes of others and to accept or reject these changes, as well as allowing them to propose alternative changes in case of a rejection. This leads to the first research question:

"Which modeling concept is needed to allow for seamless data integration in a heterogeneous software landscape?"

Previous literature research has shown the need to create such data integration model design (tough it is not an entire strategy), while comparison of selected (established and experimental)

modeling styles has allowed the selection of a specific approach (for our needs). In addition, reviewing the structure of said design model, a check-list-based procedure to create a specific design was derived, exploiting the dependencies of the sub-models to reduce design effort and error rate. (See "Structuring the Design of a Data Model" in chapter 7 for the full requirements and "Modeling Approach" in chapter 8 for a detailed evaluation.)

In short, the work regarding the first research issue only deals with the design of a data model for an integration solution, and compares various design styles and how they perform when tested with real-world use cases. It highlights the strengths and weaknesses of four different data modeling styles to determine the most flexible and reliable approach for use in complex integration solutions. In addition, by addressing known design pitfalls, time-consuming (specification) tasks and by reflecting on the structure of the design style, a creation guideline to ease the modeling was derived. However, finding a suitable data model design style alone does not guarantee the correct execution when designing a data model for integrating multiple tools (and disciplines), which brings up the next (sub)section.

9.1.2 Validating An Integration Solution Design

Since, the initial data model design has a severe impact on the later architecture of the entire integration solution, it would be advisable to have a means to check the feasibility and correctness of said design. This brings up the second research question:

"After modeling an integration solution for a specific use case, is there a generic way to validate its designed data model (and mappings)?"

Unlike the guideline on how to design a model from scratch, which originated from the initial data model design evaluation, the validation guideline had to be drafted from known design errors and past mistakes (in the prototyping). This validation check-list does not focus on the sub-models one by one, but traverses through the data model design in a different manner, by inspecting entities of lower complexity (single fields) to higher (queries for endpoints of use cases). This is done to reduce repetition of inspection steps and to find common (known) errors early on by looking for consequences of those errors. On the one hand, the procedure guarantees that all components are fully inspected (unless in case of extremely flawed designs), while detecting all previously known types of errors. On the other hand, it keeps the inspection shorter than it would be to repeat the design guideline to check through all components. Finally, by including the "Goal Validation", the inspection allows to show that the data model design not only is error-free within adherence to the specification, but also can be used for the given use cases (or allows to show shortcomings of the specification). (See "How to Validate a Data Model Instance Upfront?" in chapter 7 for the necessary content of the guideline and "Validating a Given Design " in chapter 8 for the reasoning behind the guideline.)

9.1.3 Modifying Existing Integrations Solution Designs

As prototyping of application integration solutions already implies further development and changes to a deployed installation, perhaps even a complete removal and re-creation, a data

modeling and validation guideline would be incomplete without a way to cover for changes to the data model. With change requests happening for a series of reasons and having to be able to be applied by almost all members of the integration project, this brings up the third research question:

"With reoccurring changes to the data models, how can we distinguish between valid changes and erroneous modifications?"

For the validation ("entry test") of change requests, the initial thought is to re-run the validation check from before: The result of the check either shows that the data model is still valid or that the change breaks the design. However, this is quite tedious, so it would be favorable to only run those parts of the inspection that deal with the modified parts of the data model. Therefore, running a reduced set of checks and escalating if side-effects are uncovered is proposed, while also using these mechanism to uncover changes that are purely "cosmetic" (i.e. happen inside a single data model without doing anything but renaming one or a few fields). (See "Handling Updates to the Data Model Instance" in chapter 7 for requirements of the check-list and "Handling Changes to the Design" in chapter 8 for the evaluation of the proposed solution.)

9.1.4 Result

Simply put, the suggested solution affects parts of the development process shown below (figure 3.2) by pushing forward the evaluation of the data model design and by decoupling it from the (software) development process. In addition, it formalizes the design and update process of said data models and helps to avoid and/or detect common design errors early on. This shortens the feedback loops for the (initial) data model design phase, which should allow for a faster and more precise (integration solution) data model while adhering to the criteria for a good solution:

1. ease-of-use method for deriving integration models - by providing a guideline (and an applying design style) for a specific modeling approach that
    - checks if the approach fits the challenges and
    - contains step-by-step instructions on how to create a data model design
2. robustness against later data model changes - by
    - using a data model that is designed to handle update and
    - assisting said update implementation with the same guideline
3. high rate of detection of common design errors and invalid specifications - by
    - having included checks against all common errors in the guideline and
    - using goal(use-case)-based checks to detect unfeasible designs

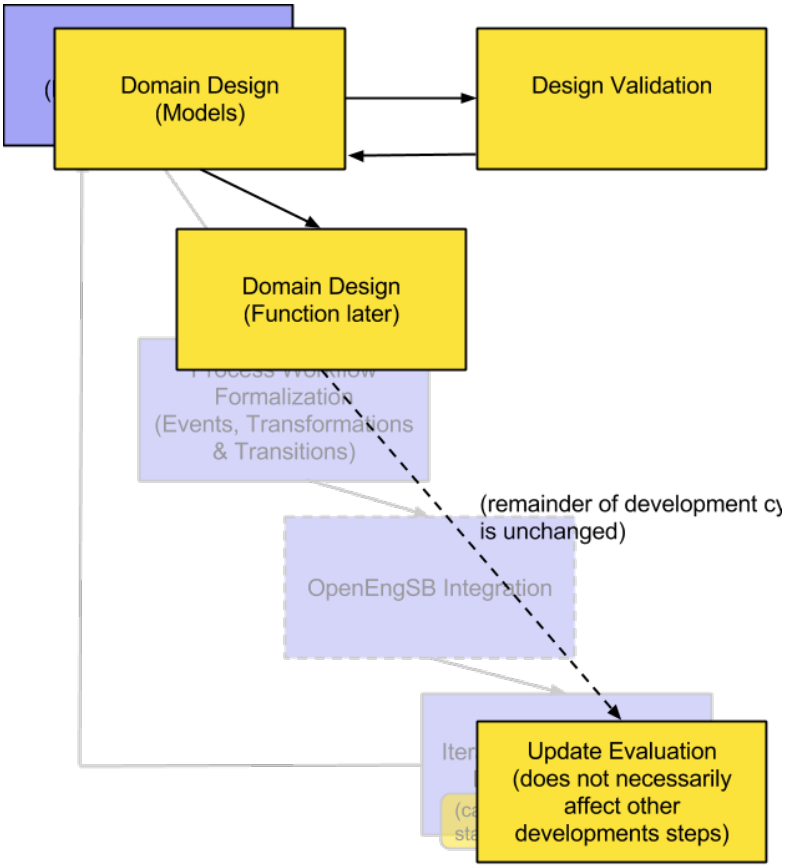


Figure 9.1: Modified development/modelling (sub)process

9.2 Limitations

Following the evaluation of this work's results are the additional insights from said evaluation, which will be used to highlight possible limitations and shortcomings of the proposed solution(s).

9.2.1 Limited Area of Applicability of the Solution

The most obvious downside of the proposed data modeling style is its relatively narrow area of applicability. The data model design style, the step-by-step guide to create such a design and its validation procedures were made with a very specific the scenario in mind: that of middle-sized to large engineering project with two or more (engineering) disciplines collaborating with each other. The basic assumption for this scenario is that there exists some common data model (even if only in the engineers' imagination so far) or multiple common data models which are used to exchange the artifacts created by each respective group. Furthermore, it is assumed that each data model is a source as well as a sink for data, with all data sources treated (almost) as equals.

While it is true that some of these assumptions do not have to be true for the design style to be applicable, once the basic scenario of a (potential) application integration project differs too much from these criteria, another modeling style might prove the a better approach. In addition, the proposed solution has "only" been verified in Automation Engineering projects, but was not tested in projects of other (related) areas, such as Automotive or Aerospace, which also can be categorized as "Software+ Engineering" [60] [33] projects (which is the original area of applicability for the OpenEngSB project).

9.2.2 Complexity and Tediousness of the Data Model Design

Another downside of the proposed solution which was already hinted at parts of this work is its high complexity. The 3-layered approach consists of tool data models (TDM), tool domain data models (TDDM) and common data models ("engineering objects"), combined with transformations (TDM to TDDM conversions) and mappings (TDM to engineering objects). As these 3 data models and two (different) conversion configurations have to be carefully modeled and maintained, the design and update guidelines can appear quite tedious. Even more, it is next to impossible to properly visualize the entire model within one diagram or table; it is feasible to show either the data models or the conversions, but not both at once. This causes the data model design's being quite hard to grasp (and maintain) for a newcomer, while people familiar with the design also should adhere to the strict guidelines, as it is easy to overlook some relations in the data model quite easily. (On the other hand, it is a model of a complex system, so oversimplifying it does not help either.)

9.2.3 Limited Scope of the Solution

The third major downside of the proposed solution is that it does not cover the entire problem. During prototyping sessions and user testing of such prototypes (mostly based on the VCDM style), a frequent complaint was the lack of an efficient and user-friendly way to modify conversion configurations. With the 3-layered data model design, updating data conversions and models is even more challenging to a user, which requires that an application integration developer is readily available to update each specific data model design. Furthermore, the data model design does not detail how the actual business logic is implemented, nor does it ease the use of a specific mechanism. While the OpenEngSB Workflows [1], which were used in the prototypes during our testing projects, worked considerably well, we cannot say whether or not this is a feasible solution for future projects.

[1] `http://openengsb.org/manual/openengsb-framework-manual/v2.5.1/html-single/openengsb-framework-manual.html#user.workflows`

CHAPTER 10

Conclusion and Future Work

This chapter summarizes the findings of chapters 6, 8 and 9 to provide a quick overview of this work's results and their possible application. In addition, suggestions for future work are derived from the known limitations of this work's proposed solution, the new applications it creates and the challenges unknown previous to this paper.

10.1 Conclusion

When designing and implementing application integration solutions that enable the exchange of engineering artifacts across discipline and tool boundaries, the initial data model design can ultimately decide the success of such an integration effort (among other risks which occur in later stages). Even with application integration projects being executed since over a decade now, there are very few established "patterns" or strategies for modeling such systems. Most of these integration scenarios can be simplified to a fixed number of tools (usually 3 to 5) all of which are each used by one discipline (such as electrical, mechanical and software engineering). The engineers of each group now have to export their respective data/work sets from the tools and send them to the other groups, so that they can adjust their own work accordingly [7]. Based on this scenario (referred to as "Engineering Object Review Cycle"), prototypic implementations using various modeling styles were tested to see how well they allowed for the covering of existing workflows and exchange data models.

The proposed solution, based on the OpenEngSB [60], was designed after reworking the design behind previously deployed working prototypes which already allowed for a data exchange across 3 disciplines. This model was again evaluated using the ATAM [43] before being used it to build a specific data model design for our test scenario. It is based on a 3-layered approach that is composed of the following models:

- The tools' data model (TDM), which is not explicitly present in the codebase of the solution, but in the configurations of parser to import the tools' data into the bus.

- The “Tool Domain Data Models“ (TDDM), which are abstractions of said tools, but aim to allow for the integration of similar tools against a single domain. The main categorization of a domain is its engineering discipline (e.g. electrical, mechanical or software), which can easily be identified by checking what an engineer aims to achieve by using a specific tool. If the overall purpose is the same for two or more tools (e.g. create a circuit design), they belong to the same domain.

- The ”highest“ or most abstract data model are the ”Engineering Objects“, which represent the common data model between two or more disciplines/domains. This common model often exists before the design of an application integration solution, but is specified only in informal ways in most cases, as the tools themselves do not support such an inter-tool exchange (hence the need for an integration solution). Note that from a technical point of view, Engineering Objects are identical to domains, but are not based on specific tools. Furthermore, they will be labelled differently to allow for a better distinction.

As already hinted, conversions between these models have to be modeled as well, with these conversions being split into two types:

1. ”Mappings“ denote the tool-to-domain and domain-to-tool conversions which are used to parse the tools’ export dumps and to create files the tools can readily import, used to bring data ”into and out of the bus“. These mappings are usually lightweight string formatting and renaming operations (splitting, concatenating and truncating) used to ”loosen up“ fields that contain multiple pieces of information or are simply named in an unintuitive matter.

2. ”Transformations“ denote the domain-to-domain conversions (including ”Engineering Objects“) that are used to transfer data model sets “inside the bus”. These transformations can look quite similar to mappings, but usually they contain more complex operations, such as calculating key fields, multi-level table look-ups or 1:n/n:1 mappings of data sets, i.e. when splitting a data instance into two or more isolated entities and keeping a link between them.

As a design style alone is not enough to guarantee a correct data model for a given project, further “before” and “after” guidelines were created to simplify the use of this style. The “before guideline” consists of 7 steps to “build” such a data model design in a controlled way. These seven phases are as follows:

1. Create a description of each tool data model.
2. Extract a more abstract tool domain data model from each of the tool data models.
3. Merge tool domains if they are used for the some overall purpose.
4. Specify the mappings to import and export tool data into/from tool domains.
5. Design the Engineering Objects for each use case.
6. Merge Engineering Objects from different use case if they are highly similar.

7. Specify the data conversion for domain-to-domain and domain-to-Engineering-Object exchange.

The "after guideline" to the data model design style consists of a series of checks that guide the developer/designer through a full inspection of the data model design with keeping redundant checks at a minimum. This allows for a (relatively) simple evaluation of a specific design without having to implement a full prototype. The evaluation is split into 5 phases, which are as follows:

1. Key validation to check for correct references of data instances, especially across data model boundaries.
2. Field validations to check if all data fields are necessary (i.e. have a "useful" origin) and if the data models themselves can be converted using the given specifications.
3. Mapping validations are test runs of the same given conversions to ensure they are specified in enough detail to allow for an automated execution.
4. Transformation validation is performed in the same way, but deals with "internal" conversions.
5. Goal validation exploits the results of all previous phases to test whether or not the given data model design serves all known use cases (much like the ATAM validates if the core business goals are actually served by an architecture).

Likewise, a similar guideline was created for (in)validating later updates to the design. This is basically a partial repetition of the previous inspection that aims to highlight the impact of said changes.

10.2 Future Work

As this work presents a data model design style, many steps of creating such a design are still done manually. Even though the guidelines aim to reduce redundant work and free the designers/developers of some tedious tasks, there is a series of possible improvements to their proposed approach.

Conversion Configuration Editor

As the underlying technologies for the data conversions have only changed once during our recent prototyping iterations, a regular (power) user request was to support for some kind of editor. However, even if one discards usability and user-friendliness, creating a tool for the interactive modification of arbitrary data models, is a complex task, as such an editor should

- only create syntactically correct conversion configurations,
- allow the user to load the source and target data models of the conversions and
- enable the users to test the conversions with valid test samples.

Model Editor and Tool-based Support of the Guidelines

In the same manner, an editor to modify and extend existing data models could easily assist the continued use of a deployed application integration solution. Current prototypes already include the possibility to extend domain data models by non-critical fields (often used for user-specific sorting and filtering views), however, to allow for a (partial) automation of the evaluation guidelines would boost the proposed solution maintainability even further, as some of the tedious repeated checks can be automated if all data models and conversion can be accessed and valid test samples are provided.

Reviewing the Limitations of the Approach

The limitations that led to the proposed solution were accepted, as the field of Automation engineering is sufficiently large to justify the past efforts of the CDL-Flex and subsequent works of its members. However, it is quite possible that a data integration strategy as proposed in this work can be extended to other fields as well. As the lack of general data integration "patterns" is not limited to Automation Engineering, this work's results may serve as a starting point for similar approaches in other (engineering) fields.

Bibliography

[1] *Datenaustausch in der Anlagenplanung mit AutomationML: Integration von CAEX, PLCopen XML und COLLADA (VDI-Buch) (German Edition)*. Springer, 2009.

[2] Sid Adelman, Larissa Moss, and Majid Abai. *Data Strategy*. Addison-Wesley Professional, 2005.

[3] Wael Al Sarraj and Olga De Troyer. Web mashup makers for casual users: a user experiment. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pages 239–246, New York, NY, USA, 2010. ACM.

[4] Aybüke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. *Softw. Test., Verif. Reliab.*, 12(3):133–154, 2002.

[5] AutomationML e. V. c/o IAF. https://www.automationml.org/o.red.c/home.html. Accessed: 2012-07-25.

[6] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[7] S. Biffl, A. Schatten, and A. Zoitl. Integration of heterogeneous engineering environments for the automation systems lifecycle. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 576–581, 2009.

[8] Stefan Biffl, Dietmar Winkler, Reinhard Höhn, and Herbert Wetzel. Software process improvement in europe: potential of the new v-modell xt and research issues. *Software Process: Improvement and Practice*, 11(3):229–238, May 2006.

[9] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Model-driven development of complex and data-intensive integration processes. In *Model-Based Software and Data Integration*, volume 8 of *Communications in Computer and Information Science*, pages 31–42. Springer Berlin Heidelberg, 2008.

[10] Nelis Boucké, Danny Weyns, Kurt Schelfthout, and Tom Holvoet. Applying the atam to an architecture for decentralized control of a transportation system. In *Proceedings of the Second international conference on Quality of Software Architectures*, QoSA'06, pages 180–198, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Andrew Branson, Tamas Hauer, Richard McClatchey, Dmitri Rogulin, and Jetendr Shamdasani. A data model for integrating heterogeneous medical data in the health-e-child project. *CoRR*, abs/0812.2874, 2008.

[12] Bundesministerium des Innern, IT-Stab (DE). http://www.v-modell-xt.de/. Accessed: 2012-07-25.

[13] David Chappell. *Enterprise Service Bus: Theory in Practice*. O'Reilly Media, 2004.

[14] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[15] C. Chirathamjaree. A data model for heterogeneous data sources. In *e-Business Engineering, 2008. ICEBE '08. IEEE International Conference on*, pages 121–127, 2008.

[16] Methanias Colaco Junior, Manoel Gomes Mendonca, and Francisco Rodrigues. Data warehousing in an industrial software development environment. In *Proceedings of the 2009 33rd Annual IEEE Software Engineering Workshop*, SEW '09, pages 131–135, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Udi Dahan. http://www.udidahan.com/2009/12/09/clarified-cqrs/. Accessed: 2013-07-02.

[18] Asit Dan, Robert D. Johnson, and Tony Carrato. Soa service reuse by design. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, SDSOA '08, pages 25–28, New York, NY, USA, 2008. ACM.

[19] Stefan Biffl Dietmar Winkler, Richard Mordinyi. Research prototypes versus products: Lessons learned from software development processes in research projects. In *18th EuroSPI Conference*, 2011.

[20] R. Drath. Let's talk automationml what is the effort of automationml programming? In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–8, 2012.

[21] R. Drath, A. Luder, J. Peschke, and L. Hundt. Automationml - the glue for seamless automation engineering. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 616–623, 2008.

[22] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, June 1999.

[23] M.E. Fagan. Advances in software inspections. *Software Engineering, IEEE Transactions on*, SE-12(7):744–751, 1986.

[24] Hao Fan. *Investigating a Heterogeneous Data Integration Approach for Data Warehousing*. PhD thesis, School of Computer Science & Information Systems Birkbeck College, 2005.

[25] Stefan Ferber, Peter Heidl, and Peter Lutz. Reviewing product line architectures: Experience report of atam in an automotive context. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 364–382, London, UK, UK, 2002. Springer-Verlag.

[26] Martin Fowler. http://martinfowler.com/bliki/cqrs.html. Accessed: 2013-07-02.

[27] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[28] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Comput. Surv.*, 28(2):415–435, June 1996.

[29] Anthony Giordano. *Data Integration Blueprint and Modeling: Techniques for a Scalable and Sustainable Architecture*. IBM Press, 2011.

[30] Li Gong. A software architecture for open service gateways. *Internet Computing, IEEE*, 5(1):64–70, 2001.

[31] B.R. Gritton. Inter-enterprise integration x2014; moving beyond data level integration. In *OCEANS 2009, MTS/IEEE Biloxi - Marine Technology for Our Future: Global and Local Challenges*, pages 1–10, 2009.

[32] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 9–16. VLDB Endowment, 2006.

[33] Michael Handler. Semantic data integration in (software+) engineering projects. Master's thesis, Vienna University of Technology, 2010.

[34] J. Jeffrey Hanson. *Mashups: Strategies for the Modern Enterprise*. Addison-Wesley Professional, 2009.

[35] Michael Havey. *SOA Cookbook: Master SOA process architecture, modeling, and simulation in BPEL, TIBCO's BusinessWorks, and BEA's Weblogic Integration*. Packt Publishing, 2008.

[36] Carsten Hentrich and Uwe Zdun. Patterns for business object model integration in process-driven and service-oriented architectures. In *Proceedings of the 2006 conference on Pattern languages of programs*, PLoP '06, pages 23:1–23:14, New York, NY, USA, 2006. ACM.

[37] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger. *UML @ Work, Objektorientierte Modellierung mit UML 2*. dpunkt.verlag, 3. edition, 2005.

[38] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[39] IBM Coporation. How service-oriented architecture (soa) impacts your it infrastructure, 20112008.

[40] S.M.S. Islam and M. Rokonuzzaman. Adaptation of atamsm to software architectural design practices for organically growing small software companies. In *Computers and Information Technology, 2009. ICCIT '09. 12th International Conference on*, pages 488–493, 2009.

[41] Tetsuo Kamina and Tetsuo Tamai. Lightweight scalable components. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 145–154, New York, NY, USA, 2007. ACM.

[42] Erik Kamsties and H. Dieter Rombach. A framework for evaluating system and software requirements specification approaches. In *Proceedings of the International Workshop on Requirements Targeting Software and Systems Engineering*, RTSE '97, pages 203–222, London, UK, UK, 1998. Springer-Verlag.

[43] Rick Kazman, Mark Klein, and Paul Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Carnegie Mellon Uiversity, Software Engineering Institute, 2000.

[44] Sami Kollanus and Jussi Koskinen. Survey of software inspection research: 1991-2005. *Computer Science and Information Systems Reports, Working Papers WP-40, Jyväskylä University Printing House, Jyväskylä, Finland*, 39, 2007.

[45] Oliver Laitenberger. Cost-effective detection of software defects through perspective-based inspections. *Empirical Softw. Engg.*, 6(1):81–84, March 2001.

[46] Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

[47] Lei Li, Roop G. Singh, Guangzhi Zheng, Art Vandenberg, Vijay Vaishnavi, and Sham Navathe. A methodology for semantic integration of metadata in bioinformatics data sources. In *Proceedings of the 43rd annual Southeast regional conference - Volume 1*, ACM-SE 43, pages 131–136, New York, NY, USA, 2005. ACM.

[48] David S. Linthicum. *Enterprise Application Integration*. Addison-Wesley Professional, 1999.

[49] Yan Liu, Xin Liang, Lingzhi Xu, Mark Staples, and Liming Zhu. Using architecture integration patterns to compose enterprise mashups. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 111–120, 2009.

[50] F. Mata, A. Pimentel, and Sergio Zepeda. Integration of heterogeneous data models: A mashup for electronic commerce. In *Proceedings of the 2010 IEEE Electronics, Robotics and Automotive Mechanics Conference*, CERMA '10, pages 40–44, Washington, DC, USA, 2010. IEEE Computer Society.

[51] P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 227–238, 2003.

[52] B. Meyer. Reusability: The case for object-oriented design. *Software, IEEE*, 4(2):50–64, 1987.

[53] Microsoft Corporation. *Integration Patterns (Patterns & Practices)*. Microsoft Press, 2004.

[54] H. Mili, F. Mili, and A. Mili. Reusing software: issues and research directions. *Software Engineering, IEEE Transactions on*, 21(6):528–562, 1995.

[55] Michael Mireku Kwakye, Iluju Kiringa, and Herna L Viktor. Merging multidimensional data models: A practical approach for schema and data instances. In *DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 100–107, 2013.

[56] T. Moser, R. Mordinyi, D. Winkler, and S. Biffl. Engineering project management using the engineering cockpit: A collaboration platform for project managers and engineers. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 579–584, 2011.

[57] T.R. Gopalakrishnan Nair and V. Suma. A paradigm for metric based inspection process for enhancing defect management. *SIGSOFT Softw. Eng. Notes*, 35(3):1–, May 2010.

[58] Heiko Paulheim. *Ontology-based Application Integration*. Springer, 2011.

[59] Christian Pena, Maria Cecilia Bastarrica, and Daniel Perovich. Atam-hw: Extending atam for explicitly incorporating hardware-related trade-off decisions. In *Proceedings of the 2010 XXIX International Conference of the Chilean Computer Science Society*, SCCC '10, pages 119–123, Washington, DC, USA, 2010. IEEE Computer Society.

[60] Andreas Pieber. Flexible engineering environment integration for (software+) development teams. Master's thesis, UT Vienna, 2010.

[61] Yrjo Raivio, Sakari Luukkainen, and Antero Juntunen. Open telco: a new business potential. In *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*, Mobility '09, pages 2:1–2:6, New York, NY, USA, 2009. ACM.

[62] A. Raza, H. Abbas, L. Yngstrom, and A. Hemani. Security characterization for evaluation of software architectures using atam. In *Information and Communication Technologies, 2009. ICICT '09. International Conference on*, pages 241–246, 2009.

[63] April Reeve. *Managing Data in Motion: Data Integration Best Practice Techniques and Technologies (The Morgan Kaufmann Series on Business Intelligence)*. Morgan Kaufmann, 2013.

[64] Ville Reijonen, Johannes Koskinen, and Ilkka Haikala. Experiences from scenario-based architecture evaluations with atam. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 214–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[65] Guido Schmutz, Daniel Liebhart, and Peter Welkenbach. *Service Oriented Architecture: An Integration Blueprint*. Packt Publishing, 2010.

[66] Alexander Schwinn and Joachim Schelp. Design patterns for data integration. *J. Enterprise Inf. Management*, 18(4):471–482, 2005.

[67] R.W. Selby. Enabling reuse-based software development of large-scale systems. *Software Engineering, IEEE Transactions on*, 31(6):495–510, 2005.

[68] Y. Sharma, R. Nasri, and K. Askand. Building a data warehousing infrastructure based on service oriented architecture. In *Cloud Computing Technologies, Applications and Management (ICCCTAM), 2012 International Conference on*, pages 82–87, 2012.

[69] Amy Shuen. *Web 2.0: A Strategy Guide: Business thinking and strategies behind successful Web 2.0 implementations*. O'Reilly Media, 2008.

[70] Steve Smith. http://programmer.97things.oreilly.com/wiki/index.php/dont́_repeat_yourself. Accessed: 2013-07-02.

[71] Sun Sup So, Yongseop Lim, Sung Deok Cha, and Yong Rae Kwon. An empirical study on software error detection: Voting, instrumentation, and fagan inspection. In *APSEC*, pages 345–. IEEE Computer Society, 1995.

[72] Wikan Sunindyo, Thomas Moser, Dietmar Winkler, and Richard Mordinyi. Project progress and risk monitoring in automation systems engineering. In Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann, editors, *Software Quality. Increasing Value in Software and Systems Development*, volume 133 of *Lecture Notes in Business Information Processing*, pages 30–54. Springer Berlin Heidelberg, 2013.

[73] ThoughtWorks. http://docs.seleniumhq.org/projects/ide/. Accessed: 2013-07-02.

[74] Tijs Van Der Storm. Generic feature-based software composition. In *Proceedings of the 6th international conference on Software composition*, SC'07, pages 66–80, Berlin, Heidelberg, 2007. Springer-Verlag.

[75] Verein Deutscher Ingenieure e.V. . http://www.vdi.eu/engineering/vdi-guidelines/. Accessed: 2012-07-25.

[76] F. Waltersdorfer, T. Moser, A. Zoitl, and S. Biffl. Version management and conflict detection across heterogeneous engineering data models. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 928–935, 2010.

[77] Florian Waltersdorfer. Data integration in software-intensive systems engineering - the engineering data base concept and applications, 2010.

[78] Norman Wilde, Sharon Simmons, Michael Pressel, and Joseph Vandeville. Understanding features in soa: some experiences from distributed systems. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, SDSOA '08, pages 59–62, New York, NY, USA, 2008. ACM.

[79] R. Winter. A current and future role of data warehousing in corporate application architecture. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 8 - Volume 8*, HICSS '01, pages 8014–, Washington, DC, USA, 2001. IEEE Computer Society.

[80] Grey Young. http://cqrs.wordpress.com/documents/cqrs-introduction/. Accessed: 2013-07-02.

[81] Nan Zang and M.B. Rosson. What's in a mashup? and why? studying the perceptions of web-active end users. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, pages 31–38, 2008.

[82] Fengyuan Zhong. Geological data integration and sharing on the semantic level. In *Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on*, pages 369–372, 2012.

List of Figures