Abstract

To master systems complexity, their industrial development requires specialized heterogeneous views and techniques and – correspondingly – engineering tools. These views generally cover only parts of the system under development, and critical development defects often occur at the gaps between them. To successfully achieve an integration that bridges these gaps, we must tackle it both from the methodical as well as from the tooling sides. The former requires answers to questions like: What are the views provided by the tools? How are they related and extended to achieve consistency or to form new views? – while the latter requires answers to: How are views extracted from the tools? How are they composed and provided to the user? Our approach, suitable for incremental integration, is demonstrated in the tool integration framework TOOLNET.

1 Introduction

State of the art systems engineering practice involves a variety of heterogeneous software tools covering a specific engineering process. As the introduction of new CASE/CAE tools is generally restricted by non-technical constraints (e.g. cost of introduction, availability or user acceptance), processes are geared to the tools available. This results in tool customization and integration to optimally support engineering processes.

As most of the tools only cover a subset of the process activities, the entire model of the system under development is split up into partial, tool-supported models, that are restricted system views. These generally overlap and depend on each other: They may contain related elements of the entire model, overlap by sharing common elements or be related through process activities.

The related views have to be also technically integrated. Otherwise, inconsistencies can arise, either caused by redundant partial models, if views implicitly overlap or by unrelated partial models, if implicit dependencies between views are not captured explicitly by a tool.

Thus, tool integration has to deal with inconsistencies caused by a mismatch between the logical, overall model – existing only implicitly – and the technical partial models – existing explicitly in the tools – of the system under development. We consider tool integration not only being a technical issue – in terms of exchange formats or adapters to allow information sharing – but more importantly as a methodical issue – requiring to understand the commonalities and dependencies between the tools’ models. Nevertheless, model integration literature mostly concentrates on technical details (Section 5). We can identify a gap between the state of the art in academia and in practice where additional problems are encountered:

Heterogeneity – Usually, tools to be integrated are at various abstraction levels, belong to various development phases (e.g. requirements, design) or address various domains (e.g. mechanical, electrical). The models defined in these tools have different levels of precision: Ranging from structured natural language text (e.g. requirements) up to formal specifications. Thus, import or export of models and automatic rule-based integration are hard to achieve.

Ambiguity – Often, modelling constructs of a tool are too general to offer good support for a particular project. Thus, the model developed within an integrated tool, referred to as product artifact (PA), is weakly defined or too complex. A tool’s metamodel of data and operations, referred to as tool model (TM), hardly captures all the modelling constructs that would be specific to the developed class of systems. Project specific conventions often are superimposed on such metamodels.

Opacity of tools – Most of the tools only offer restricted access to their internal models and this causes serious technical problems that limit the integration depth a tool can support.
We assume that above issues are, and will be on medium term, inherent in practice. Thus, our approach is to incrementally capture and analyse integration needs in terms of integration concerns. Moreover, we aim at covering an integration problem with a set of integration scenarios which characterize the semantics of the integration for particular purposes.

In Section 2, we present a set of relevant integration concerns that frequently appear in a systems engineering environment. In Section 3, we introduce our layered tools integration framework and discuss the problems addressed by each layer and those which appear when crossing the layers. Section 4 presents how our solution systematically resolves the integration problems from Section 2. Before drawing a conclusion, Section 5 provides an overview of related work.

2 Tool Integration in Practice

A systems engineering “digital” environment (SEDE) typically contains a set of tools, e.g., predefined by acquirer-supplier relationships or required for certification procedures. As explained in Section 4, we consider an avionics engineering process phase using a configuration of DOORS and SCADE to outline an integration of requirements with specifications. While DOORS contains requirements in terms of structured text, SCADE incorporates formal system specification by means of functional node networks. A typical certification constraint requires traceability: Each DOORS functional requirement (requirements specification) should be covered by a formal SCADE functional node (design). This may raise three essential needs for integration, referred to as integration concerns (IC):

Model associations – To ensure traceability between requirements and their counterpart functions, e.g. imposed by the verification procedures of standards for safety-critical software in avionics [18], requirements of DOORS models must be explicitly linked to nodes of SCADE models.

Model extension – To enable change control on requirements and design, e.g. as required by the configuration management process of the CMMI level 2, revision dates of DOORS requirements and SCADE nodes have to be available.

Model restriction – To formalize the change control process the revision date of a functional requirement must be earlier than the revision date of the corresponding SCADE node.

Above concerns are ensuring certification – tracing requirements, recording revisions, and checking revision dependencies – and require a specific part of the system description, consisting of the models of DOORS and SCADE enhanced with associations between them, revision dates and consistency conditions.

2.1 Integration Scenarios

From the previous examples of ICs, we can derive functional cases of integration, referred to as integration scenarios (IS) and characterized as follows:

Link models – In order to express associations between models, we need explicit links between their elements. These links can be used to capture loose relationships like traceability between different artefacts – e.g. implementation, derivation, or allocation of requirements.

Extend model(s) – In order to capture semantic information about the models, we need to add new classes of model elements or to enhance existing model elements with additional information (e.g. revision date).

Restrict model(s) – Applying consistency conditions to integrated models also needs us to separately impose constraints on their probably isolated constituents.

2.2 Practical Issues of Integration

In addition to the mentioned integration scenarios, an industrial tool integration solution also implies some requirements and restrictions:

A posteriori integration – We have a fixed landscape of tools, and thus we need to achieve the integration of existing models. The integration of these tools is often not focused by their manufacturers. These tools are often not meant to be integrated at all.

Variable integration depth – The depth of integration should be adaptable to the needs of the engineers and the capabilities of integrated tools. On long term we aim to integrate “as much as possible” both from the technical and the methodical point of view. But on medium term we assume to benefit from light-weight integration by regarding existing tools as a central aspect of integration and building around our integration solution.

Incremental integration – Changing integration concerns positively correlated with an evolving information space or modelling and tool landscape may on principle prohibit an inflexible integration solution. Already hinted in [19], integration is usually as fluctuating as the integrated information itself. Thus, regular and incremental enrichment of the integration solution is substantial for its continuous success. Section 5.3 describes how our approach takes this issue into account.
In Section 3, we focus on the implementation of the mentioned ISs. They require engineers to create parts of the entire logical product model which virtually spans over multiple tools. As shown in Figure 1, there is a gap between a low-level tool landscape and the high-level, virtual model of the entire product. The biggest difficulty in implementing the scenarios presented above is to bridge the logical and the tools layers. We combine bottom-up tool-specific modelling and top-down integrated product or information modelling.

3 4-layered Tool Integration Framework

In order to integrate an a priori set of tools we enhance the (implicit) models of these tools by additional semantic information, e. g. associations, and consistency conditions. We model the required ISs by partially expressing or approximating a virtual product model in terms of views, further referred to as integration views (IV). This requires an extraction of the implicit models available in the integrated tools and to capture the de facto information already present in the development process. Thus, one cornerstone of our approach is to reasonably balance bottom-up and top-down modelling to achieve maximal leverage of the exiting tools.

We use the term *virtual product model (VPM)* synonymously to integrated or common product model, thereby meaning a complete logical description of the entire system under development. By the term “virtual” we emphasize not to achieve a complete representation of the product model by means of information or data.

In achieving the integration one needs to manage problems ranging from accessing the internal tool data to expressing logical dependencies in terms of the integrated models. In this section we present these issues in a layered integration architecture which comprises aspects of integration specific to tools, models and the product.

3.1 The Tools Layer \((L_T)\)

On of our main assumptions is the *a priori* existence of the engineering tools, that need to be integrated. The tool landscape is fixed, its replacement is not an option and the needs for integration are hardly anticipated adequately by the initial tool manufacturer.

These tools belong to different engineering domains (e.g., electrical, mechanical) and their purpose is to optimize a particular set of domain-specific engineering tasks. In order to do so, each tool offers a standard notation and methodology which supports a standard set of activities that are the most effective ones for its domain. Due to the heterogeneity mentioned in Section 1, the only similarity left between these tools is their use within one project. Often, the PAs managed by a tool are technically hidden behind its user interface. The decision of opening a COTS tool towards interoperability mostly falls on its manufacturer’s side.

There is a conceptual gap between the VPM and the tools used for its development. We consider it to be translated into the modelling constructs offered by the tools. But high-level information necessary for recovering the VPM is either implicitly present in these tools or only in the heads of engineers.

From the perspective of interaction between engineers and tools, integration should be part of the tools encapsulated by an add-on or integration facility which can be used or ignored. Depending on the tools and the purpose defined by an integration view, the integration facilities should be seamlessly accessible through the standard user interface of the integrated tool: we aim at both the integration of tools and the integration of models. For an engineer, as far as the integration is concerned, it would be ideal to work within a single tool which provides seamless access to the whole information space.

Due to the importance of tools and the problems raised by their integration, our integration framework has an extra layer for them. As shown in Figure 2, the bottom layer contains the systems engineering tools that exist prior to integration.

3.2 The explicit Model Layer \((L_M)\)

This layer aims at the abstraction from technical details of tool integration by making hidden models explicit. Here, models of different tools are expressed in a single notation in order to eliminate their syntactic heterogeneity. It contains explicit PAs which can be accessed by the integration environment. Depending on the information provided by this layer, one can or cannot implement a particular integration scenario. It furthermore determines an important part of our pragmatic philosophy: “achieving feasible integration by taking optimum advantage of currently available potentialities of the tools”.

Each model at this level is in fact a projection of the (hidden) tool models. From the integration point of view we
can consider that this level abstracts the tools as a set of uniformly described models. The model layer is not only an abstraction of the tool layer due to technical accessibility of information but also due to methodical reasons. This abstraction allows for dropping specific aspects from the model layer, e.g. layout information contained in graphical representations or internal management data in the tools layer. Thus, the model layer should represent an “intersection” of models at the logical layer and the tools layer.

3.3 The Integration Layer ($L_I$)

This layer is responsible for the implementation of the specific integration scenarios by making explicit all interesting elements of the product model and the dependencies between them.

While the realisation of an entire product model for a complex product is not feasible, at this layer we construct different views over the product model which cross the logical boundaries of individual tools. Thus, we approximate the product model by a set of projections. Each integration view captures relevant aspects of the integrated product model which are necessary to address an IC. An integration view contains: projections of the explicit models $E_i$, additional information, referred to as metadata (MD), which is not comprised in any tool and a set of conditions on the model projections and the MD. In an integration view, MD and consistency conditions glue together the integrated models in order to implement a particular integration concern. MD characterizes a specific integration concern by capturing gaps between the tools models and mostly needs to be supplied from the exterior.

3.4 The Logical Layer ($L_L$)

The logical layer contains an overall view of the product. Important gap-filling information may be derived from the knowledge in this layer and from a combined usage of the tools. Questions like “How and why is a design element connected with a requirement?” or “What is the structure of the product from a specific point of view?” may be answered here. $L_L$ potentially contains all information that stakeholders know about the product. Usually, there is no single person who has all the information at $L_L$, instead it is spread across several engineers or teams.

3.5 Crossing the Layers

Now, we proceed in regarding the framework from a methodological perspective. We approach model and tool integration in terms of our four layers from both directions, bottom-up and top-down. Technical tool integration, the extraction of models and the interaction with the user is realised in a bottom-up manner. The definition of integration views, consistency conditions and expressing these views in terms of tools’ extracted models requires a top-down and project specific approach:

1. **Identify integration concerns** – The engineer identifies needs for integration, i.e. an integration concern. This step is completely performed at the logical layer since the integration concerns address parts of the integrated product model. In this phase, the problem is defined in a declarative manner, exclusively in terms of the product model and without considering the tools used.

2. **Design integration view** – The engineer translates the integration concern in terms of integration scenarios to the available models and captures the additional types of information for the integration view to be built. The engineer decides whether the integration concern can be implemented or not by specifying the set of accessible tool model elements.

3. **Incrementally construct integration view** – The engineer constructs an integration view in detail by making additional data and conditions explicit. This view defines the semantics of its integration concern and is realized incrementally. For each increment the engineer needs to specify which model elements are related, what information should be added and how a relationship could be expressed in terms of available data.

Once, the views are built, engineers can use them extensively to answer integration questions by starting from a specific tool model element. E.g. a valid question to
requirements and design traceability would be: Which requirement is implemented by a design element? Answering this question requires the engineer to analyze bottom-up and go from a model element to the logical layer:

1. Identify model element – The engineer needs to answer a specific integration question related to a specific tool model element.

2. Answer “integration question” – The engineer can query the integration layer to get the specific integration views which comprise the selected element. Once, such a view is found, it provides answers to the questions related to the integration concern.

The integration layer is responsible for realising the junction of bottom-up and top-down approaches. Due to the bottom-up approach, the integration layer is provided with all the necessary tool specific information. Its main responsibility is to leverage the existent tools up to the level which enables integration from a logical point of view. The top-down approach offers an interpretation of this information, which depends on the particular purpose and the logical model.

4 Case Study

We revisit our example from Section 2 to show the support of integration concerns. We restrict our example on two integrated tools. Given the technical infrastructure introduced later in Section 4.2, it is relatively easy to widen our light-weight way of integration to the scope of several tools.

4.1 Implementing Concerns through Views

The tools introduced in Section 2 support various activities: requirements management and software specification. The models of DOORS and SCADE are both logically and from a data point of view rather different. DOORS requirements are often organized according to requirements formatting rules and functional structures resulting from previous analysis steps. In contrast, SCADE offers a formal specification language for describing synchronous dataflow. Hence, its structure of data mirrors results from physical or technical analysis steps. However, some structural and organisational aspects as well as representational aspects of data structures are well combinable in terms of information management.

Figure 3 introduces partial views of the general metamodels of these tools. These views include modelling constructs used in DOORS (projects, folders, modules, requirements objects with system or user defined attributes) and in SCADE (projects, specifications, type definitions, constant definitions, global variables, functional nodes with inputs, outputs and systems of equations). We show three possible examples of meaningful relationships between DOORS as an early phase engineering tool and SCADE as a specification and implementation tool.

In the paragraphs a) to c), we describe the implementation of three integration scenarios supporting the mentioned integration concerns:

a) Link models for traceability In a general sense traceability is a chronological interrelation of entities developed across multiple project phases. Traceability between DOORS requirements and SCADE nodes refers to the ability to determine the requirement that results in a part of the system’s specification. In an integrated product model, traceability relations are described through simple associations between model entities. The DOORS and SCADE models are developed independently. We capture traceability information through a view which relates single model elements from the two models. Here, traceability is a trivial relationship where there is no need for additional information and conditions always evaluate to true. An integration increment is represented in Figure 3 and consists of a simple link which associates the related model elements.

b) Extend models to support change Changes of an artefact can motivate subsequent changes of dependent system artefacts. In order to detect this, engineers need to perform reviews by checking items, e. g. requirements, which depend on the changed artefact. As most of the system artefacts developed must satisfy a specific requirement, changes of them often affect subsequently developed artefacts.

We want to extend on a) in order to enable process support for change reviews. Thus, we need dates of the last changes to requirements, i.e. the attribute last_changed_on of DOORS elements, and additional information about the date of the last review performed (Figure 3b). The constraint would be: "last_changed_on earlierThan last_review_date".

c) Restrict models to enforce project conventions System development with multiple independent tools requires strong project discipline, which can be expressed through project conventions which cross tools’ boundaries. A naming convention for related artefacts is a suitable example: The names of the model elements developed in different tools and which refer to the same artefacts of the system under development should be similar in order to enable their identification.

In our case study, each SCADE node should be traceable to a DOORS requirement object which describes the
functionality implemented by the node. We define an integration view which associates the corresponding node to each requirement. Furthermore, in order to enforce naming conventions, each integration increment should consider the name of the SCADE node (name) and the title of the corresponding requirement (title) (Figure 3c). Project-specific conventions can then be expressed as: "name substringOf title".

4.2 The TOOLNET Integration Framework

In Figure 4 we present the logical architecture of the tool integration framework prototype TOOLNET. TOOLNET as a technical infrastructure covers three layers of the proposed integration framework.

At $L_T$ TOOLNET contains a set of tool add-ons (technically: plug-ins) which enable the communication between the integration framework and the integrated tools. Depending on the tool’s depth of integration, the add-on facilitates different types of communication and thus enables different functionality for the upper layers.

The hidden models of integrated tools are made explicit at $L_M$. Together with the additional information, definable for each model element, it constitutes the model abstraction layer (MAL).

These explicit models at the models layer are used in the definition of integration views at $L_I$. In the current TOOLNET prototype, we make use of MOF [16] and Eclipse technology [6] for the description of the tools’ metamodels, the integration views and different types of mappings.

Deploying TOOLNET in a SEDE leverages both the original tool functionality and, by utilizing tool services like mentioned above, a large variety of integration functionality. The pragmatic intention behind defining integration views is to approximate integration and integrated product models through scenarios and aim on supporting specific integration concerns. Instead of immediately achieving a complete integrated product model, we can incrementally improve the approximation and vary the depth of integration.

4.3 Discussion

The concept of mixing top-down and bottom-up integration is facilitated by the model layer, which translates a tool’s language to the language of integration views. Technically, this is done by a “model abstraction layer” (MAL). As the MAL only extracts bottom-up what can be exported via the tool’s interfaces, we cannot avoid a logical boundary of identifiable relationships and functionality within the integration layer. The top-down part consists of conceptualizing a set of integration views which the extracted or additionally created information can be mapped to.

We are looking forward to further investigate the mentioned problems and to delivering a more elaborate example and focus on the explained concept of incremental integration.
5 Related Work

In Figure 5 we depict the related approaches we investigated and classify them according to their coverage of our four layers, their time and depth of integration and their assumptions to the openness of tools. In this section we additionally classify them into user-oriented, model-based, consistency and technical approaches.

User-oriented approaches The MATra framework [14] enables traceability and consistency between highly heterogeneous engineering artefacts. Data is exported from tools to a workspace and linked to represent the whole system in terms of an integrated model for avionics computing systems. In comparison with this approach, we do not restrict to one comprehensive model but provide several views that approximate it in a scalable manner.

Altheide [1] aims at method integration. UML descriptions of development methods – consisting of coarse process artefacts, their methodical interrelationships (artefact model) and their internal structure (object model) – are used to vertically integrate tool information into the development process. A view in our general integration layer can be seen as a container for such a method description and, thus, can support process integration concerns.

Another idea is to combine process, product and tool integration in a three-layered integration architecture [9]. Regarding a multi-perspective integrated product model, this approach differs from ours in that it prefers tools staying in the background as services behind a unified user interface.

The n-dim approach [19] shows elaborate concepts of tool integration and information management. From this perspective, n-dim has much in common with our approach. As opposed to considering tools as being solely legacy, we consider adopted “commercial-of-the-shelf” (COTS) or in-house tools.

Telelogic DOORS [20], a requirements management system, provides loose integration capabilities such as simple traceability and the import of documents from other tools, e.g. Esterel SCADE [8], a tool used for specification and implementation of embedded software. However, DOORS lacks an advanced integration capability such as imposing consistency over the integrated models.

Model-based development As a tightly integrated environment prototype, AutoFOCUS [4] operates over an integrated product model. It especially addresses the interactive transition from requirements engineering to software design and specification. Focusing vertical integration, our architecture provides means to technically implement an a posteriori tool integration bridging the gaps between both models and tools.

The development of new tool chains can start by imposing a common metamodel on the modelling languages used in the tools. Tools have to be developed according to this metamodel (i.e. they are integrated top-down sharing the same ontology). Being attracted to this idea, TOPCASED [5] is a framework for metamodel-based building of editors for modelling activities in software development. As opposed to these integration approaches, we assume that the construction of the off-the-shelf tools can not be influenced in the medium term towards providing support for model-based integration.

Consistency frameworks Whereas consistency seems to be only one aspect of the work mentioned, xlinkit [15] specifically aims on this issue. Consistency is defined indirectly and declarative via rules. Inconsistencies identified during batch-wise checks are expressed through links between erroneous elements of homogeneous XML documents (exported tools’ models). Due to the difficulties in declaratively addressing elements and tracing back from XML identification schemes to original tools’ models, we provide an interactive, direct definition of relevant links in the tools.

Technical approaches Model transformation based on triple graph grammars [12] are used for tools integration. As a sound formal method for the generation of target models and corresponding consistency mappings, the technical problem how to make tools’ models explicit is not entirely
solved.

A model-driven and architecture-based approach for the incremental and cost-efficient development of software components (tool adapters and integration components) for posterior tool integration is proposed by [2], aiming a point-to-point, tightly coupled integration paradigm. We technically aim a service-based framework for mediated integration and need the access to the tools’ models to be highly scalable.

6 Conclusion

The presented approach describes the incremental integration of heterogeneous systems engineering tools, by methodically explicating the views of these tools, as well as enhancing, linking, and restricting them to approximate an overall (logical) model of the system as required by the development process.

An optimally deep integration can be achieved by providing a product model as a common repository of product data, with specialized tools accessing it to provide specific views as well as construction and analysis functionalities. In a homogeneous, academic context, tool frameworks like [4] or [13] support such an approach.

As an intermediate goal, here, a more pragmatic form of integration is introduced, addressing processes based on a heterogeneous tool chain: methodically, elements of the (implicit) models provided by the tools are embedded into views of an integrated logical model based on specific integration functionalities required by the development process; technically, an infrastructure – at the model layer – is constructed to access – at the tool layer – the implicit models, provide – at the integration layer – model extensions including linking and restriction, and offer construction/analysis functionalities for the extended views. Thus, the logical layer is only approximated by the integration layer, allowing an incremental approach to gradually scale up to a complete tool landscape.

The feasibility of such an approach has been demonstrated by the integration of DOORS and SCADE into the TOOLNET framework, especially addressing the traceability of requirements. These promising results immediately lead to further research and development issues – including a generic “tool-bus” supporting transparent access to implicit models, minimization of user-indirections for extended views, and finally support for incremental transition to deep integration – to be addressed in future work.

References

