# Workflow-Driven Tool Integration Using Model Transformations *

András Balogh[3], Gábor Bergmann[1], György Csertán[3], László Gönczy[1], Ákos Horváth[1], István Majzik[1], András Pataricza[1], Balázs Polgár[1], István Ráth[1], Dániel Varró[1], and Gergely Varró[2]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{bergmann,gonczy,ahorvath,majzik,pataric,polgar,rath,varro}@mit.bme.hu
[2] Department of Computer Science and Information Theory,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
gervarro@cs.bme.hu
[3] OptxWare Research and Development LLC,
H-1137 Katona J. u. 39.
{andras.balogh,gyorgy.csertan}@optxware.com

**Abstract.** The design of safety-critical systems and business-critical services necessitates to coordinate between a large variety of tools used in different phases of the development process. As certification frequently prescribes to achieve justified compliance with regulations of authorities, integrated tool chain should strictly adhere to the development process itself. In order to manage complexity, we follow a model-driven approach where the development process is captured using a precise domain-specific modeling language. Each individual step within this process is represented transparently as a service. Moreover, to carry out individual tasks, systems engineers are guided by semi-automated transformation steps and well-formedness constraint checking. Both of them are formalized by graph patterns and graph transformation rules as provided by the VIATRA2 framework. In our prototype implementation, we use the popular JBPM workflow engine as orchestration means between different design and verification tools. We also give some insights how this tool integration approach was applied in recent projects.

## 1 Introduction

Complex development projects, especially, in the field of safety-critical systems, necessitate the use of a multitude of software tools throughout the entire life-cycle of the system under design for requirements elicitation, design, implementation, verification and validation as well as change management activities.

However, in order to ensure safety, the verification of tool output is mandated by industrial certification standards (like DO-178B [1] for avionics systems), which requires

* This work was partially supported by the European Union as part of the MOGENTES (STREP-216679), the DIANA (AERO1-030985) and the SENSORIA (IST-3-016004) projects.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 224–248, 2010.

enormous efforts. Software tool qualification aims at reducing or even eliminating such efforts to obtain certification credit for the tool itself by ensuring that a tool always produces deterministic and correct output.

Standards differentiate between *verification tools* that cannot introduce errors but may fail to detect them, and *development tools* whose output is part of the critical system and thus can introduce errors. According to the standard, a development tool needs to be qualified to at least the same level of scrutiny as the application it is used to develop. The main functionality of a software development tool is thus to correctly and deterministically transform an input artifact into output.

Unfortunately, the qualification of software tools is extremely costly, even minor changes would require re-certification efforts [2] resulting in several man-years of work. Furthermore, qualified tools are almost exclusively relying upon closed, internal technologies of a company [3], without using external components, as vendors of safety-critical tools are unable to control the development of external components with the level of preciseness required by certification standards. Finally, the tool integration costs for building a uniform tool chain can frequently exceed the total costs of the individual tools themselves.

The extreme costs of certification and tool qualification are largely due to the fact that integration between different tools is carried out in an ad hoc way in the industry [4]. It is still a very common scenario that the output of one tool is ported manually to serve as the input of another tool. Moreover, the chaining of different tools is completely decoupled from the rigorous development processes necessitated by the standards.

In the current paper, we propose to use model transformation services organized into complex *model transformation chains*. These transformation chains are closely aligned with the designated development process as *driven by precise workflow models* where workflow activities comprise of individual development steps carried out by some tool, which creates some output artifacts (models and code, configuration files) from some input artifacts. Moreover, each step in a tool integration chain can be hierarchically refined later on by using workflows for capturing the main steps of individual development tools.

Formalizing design processes with workflow allows formal adherence checks with certification guidelines. Moreover, workflow-dirven tool integration aligns development and V&V toolchains with the actual development process itself.

Individual development steps of the transformation chain are treated as black-box components, where functionalities carried out by the tool are precisely captured by *contracts formalized as graph patterns*. This black-box approach enables that both automated and user-driven development steps can be integrated in a uniform way to the tool chain. Furthermore, *automated* tool integration or development *steps can be captured by model transformations* formalized by means of graph transformation rules.

This approach has been successfully applied to developing individual tools (as in the DECOS [5] and DIANA [6] EU projects) as well as for complete tool integration chains (as in SENSORIA [7], MOGENTES [8] and GENESYS [9] projects) in the context of safety-critical systems. Further industrialization of the framework is being carried out as part of the ARTEMIS project INDEXYS [10].

The rest of the paper is structured as follows. Section 2 summarizes the main steps and challenges of tool integration scenarios. In Sec. 3, we propose a workflow-driven approach for driving tool integration scenarios. Section 4 introduces a tool integration challenge taken from an avionics context. Section 5 discusses how graph patterns and graph transformation rules can be used in automating the development of the tool integration process. Then, in Sec. 6, we also demonstrate how such a tool integration framework has been implemented using state-of-the-art open source technologies. Finally, Section 7 discusses related work and Section 8 concludes our paper.

## 2 The Architecture of Tool Integration Chains

### 2.1 Classification of Development Activities

Complex development processes make use of a multitude of development, design, verification and documentation tools. The wide spectrum of underlying technologies, data representation formats and communication means has called for tool integration frameworks to address the need for a common underlying tool integration platform. Such middleware is typically designed to allow for various tools to be integrated as *services*, so that the integration process can be designed by concentrating on the *tasks* that are to be performed, rather than the underlying technological peculiarities.

On the conceptual level, the main functionality of each step (task) is to transform an input artifact into one or more output artifacts. This transformation view on development and tool integration tasks does not have a direct impact on the level of automation. For example, certain tasks can be either (fully) *automated*, such as compiling source code from an executable model like statecharts or running a model analysis task to reveal conceptual flaws in the design. Other development tasks are inherently *user guided* (or user driven) where a development step is completed in close interaction with the systems engineers. User guided steps typically include those where design decisions need to be made and recorded, such as modeling. While full automation is impossible (or impractical) for user guided steps, the step itself can still be interpreted using this transformational view. Moreover, automation may still implant design intelligence into such tools be performing on-the-fly validation of certain design constraints, which can reduce costs.

Development steps can also be categorized on the basis of comparing the information between the source and the target formalisms of the step.

- *Synthesis steps* (carried out by using textual or graphical editors, and even certain automated tools like schedulers, optimizers) add new information to the system under design during the completion of the step.
- *Analysis steps* (also known as verification and validation steps), on the contrary, typically abstract from existing information in order to enable checking for certain correctness properties to reveal errors in the design.
- *Derivation steps* (like code generation or model export and import with format conversion) do not add or remove information, however, they change the representation of the information.

| | Automation | Guidance |
|---|---|---|
| Analysis | Constraint checker, model analyzer | Code reviewer, test case designer |
| Synthesis | Scheduler | Modeling tool |
| Derivation | Code generator | Guided transformation |

**Fig. 1.** Process activity types

A summary of these categories are shown in Fig. 1. It is important to stress that the mode of execution and design information handling aspects are practically orthogonal, so relevant examples for all combinations can be easily given (in the table cells in Fig. 1).

### 2.2 Synthesis

Synthesis activities are typically carried out by off-the-shelf development tools such as programming environments, documentation tools, modeling editors etc. By these means, engineers create and manipulate the artefacts of the design process using manual or semi-automated procedures.

Along with design information, most (critical and expensive) *errors* (e.g. design flaws, anti-patterns) are introduced into the system-under-design in these activities. To reduce the impact of these faults, advanced tools offer checking facilities, ranging from light-weight approaches (such as syntax analysis for source code, domain-specific well-formedness checking in model editors) to more advanced features (e.g. static code analysis, model simulation by in-place execution [11]).

The quality of design synthesis output can also be improved by using semi-automated tools for design-time optimization. For instance, in embedded architecture design, automatic *schedulers* may be used to calculate the timing of a message bus communication protocol, or *resource allocation tools* may be used to optimally assign software tasks to hardware nodes.

### 2.3 Analysis

Figure 2 shows a typical approach to early analysis using model-based techniques. In model-driven development, system requirements and design are captured by high-level, visual engineering models (using popular and standardized modeling languages like UML, SysML, AADL). In critical systems, where the system under design must conform to high quality and reliability standards, early systematic formal analysis of design models plays an increasingly important role to reveal design flaws as early as possible. In general, this can be carried out by generating appropriate mathematical models by automated model transformations. Formal analysis then retrieves a list of problems, which can be fixed by refinement corrections prior to investing in manual coding for implementation. Finally, these refined models may serve as input to code generators and deployment configuration generation, which create the runnable source code of the

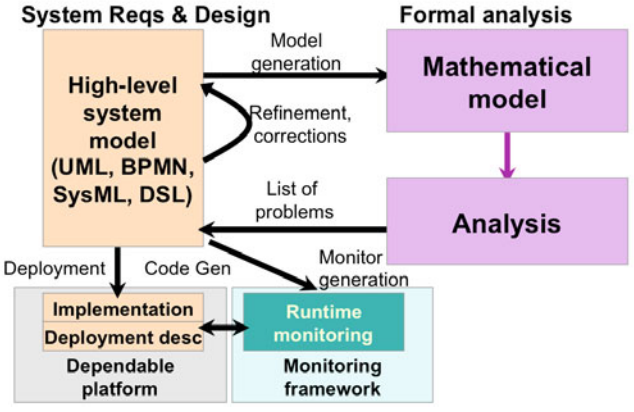



**Fig. 2.** Analysis by formal techniques

application as well as parametrize auxiliary deployment infrastructure services such as monitoring.

Analysis steps may include (i) to investigate functional correctness of the design by verifying safety or liveness properties (e.g. by model checking statecharts [12, 13]), (ii) to assess the effects of error propagation (e.g. using fault modeling and analysis techniques [14]), (iii) to evaluate non-functional characteristics of the system such as performance, availability, reliability or security (e.g. by deriving stochastic models from engineering models [15, 16, 17]) and many more. A commonality in these approaches is the extensive use of automated model transformations to carry out the abstraction necessitated by the formal analysis.

### 2.4 Derivation

Derivation steps primarily include automated code generation tasks, or to chain up several development steps by importing and exporting models in different tools.

Derivation steps can frequently be fully automated as all information required for code generation is available prior to initiating the step. Of course, such code generators may still combine the information embedded in different models to derive the designated output, or to mark the platform independent models by design decisions. Anyhow, in both cases, the actual derivation step is completed using existing information implanted by synthesis steps.

Code generators may derive the source code or the target application (see code generators of statecharts [18]), yield deployment descriptors for the target reliable platform [19, 20], or generate runtime monitors [21].

As a summary, complex tool integration frameworks should be closely aligned with development processes by taking a transformation-based view on individual development steps. Moreover, they need to simultaneously provide support to integrated automated as well as interactive, user-guided development steps where the starting point and the completion of each step needs to be precisely identified. Finally, the framework

should enable to integrate arbitrary kind of development steps including synthesis, analysis and derivation tasks.

## 3 Process-Driven Tool Integration

Based on the experience in tool integration summarized in Sec. 2, we propose an integrated approach for designing and executing tool integration processes for model driven development. As a proof-of-concept, we describe a case study developed for the DIANA [6] research project (Sections 4, 5 and 6).

By our approach, development processes are formally captured by workflow models, which (i) specify the temporal macro structure of the process as a task-oriented, hierarchic workflow model; (ii) precisely map the steps of the process to the development infrastructure, consisting of human resources (roles), tools, supporting technologies available as services and development artefacts as entities in the data repository; (iii) define high-level contracts to each step, which specify constraints that help to verify and trace the correctness of outcomes and activities.

These process models are deployed to a software infrastructure, which serves as an automated execution and monitoring platform for the development process. It provides support for running automated and user-guided activities in a distributed environment consisting of heterogeneous software tools, and a model bus-like [22] data repository.

In this section, we describe the specification language of the workflow models in the tool integration domain in detail.

### 3.1 Process Metamodel

**Macro structure.** For the specification of the temporal macro structure of development processes, we follow the notions and concepts of well known process description languages such as BPMN or XPDL. As the metamodel in Fig. 3 shows, processes are constructed from workflow `steps` (corresponding to distinct activities carried out during development), and may use standard control flow features such as sequences (`ProcessNode.next`), concurrency (fork-join) and decision points.

More advanced constructs, such as waiting states are intentionally omitted from this language, since this is intended to be a very high level description, where only the order (precedence or concurrence) of activities is important; for execution, this language is mapped to a lower level jPDL representation which may be customized and augmented with more advanced behavioral properties.

**Hierarchy.** It is important to stress the *hierarchical* nature of the process description: through the `Step.subNodes` relation, workflow steps may be embedded into each other to create a hierarchical breakdown. This allows the process designer to map the ”birds-eye-view” structure development processes (such as phases and iterations) to our language; additionally, it supports ”drill-up-drill-down”-style navigation through a complicated workflow, which is important to reduce design complexity for large-scale processes.

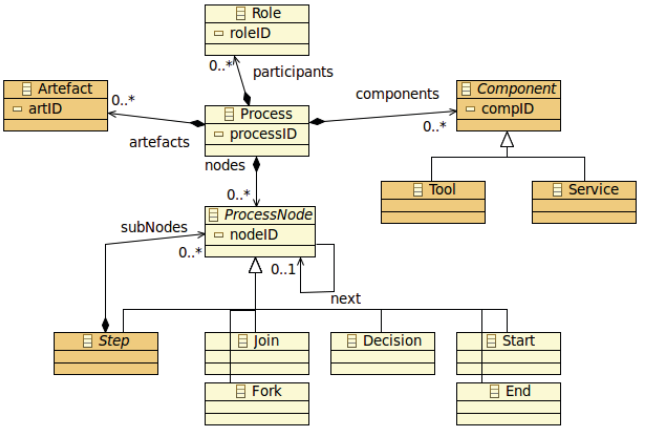



**Fig. 3.** Integration process macro structure metamodel

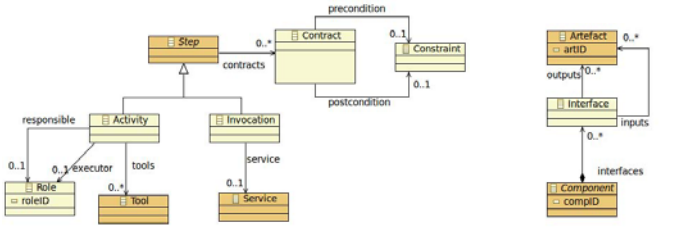
**Development infrastructure.** Besides the causality breakdown, the language features the following notions of the development infrastructure:

- `Artefacts` represent data structures of the process (e.g. documentation, models, code, generated files, metadata, traceability records).
- `Roles` correspond to the participants of the process, humans and external entities who are involved in executing the user-guided activities.
- `Components` are either `Tools` or `Services` which are used by the participants or invoked automatically during development, to complete a specific step.

These concepts enable the precise mapping of the development workflow to the actual execution infrastructure.

**Mapping to the execution infrastructure.** As shown in Fig. 4, our language can be used to precisely specify the execution of the development process. During process modeling, workflow steps are first assigned into the `Activity` or `Invocation` categories, depending on the type of interaction (activities are user-guided while invocations are automated). For user-guided tasks, the language adopts the basic role-based assignment method (taken from BPMN), and optionally supports two basic types of relations (`responsible` and `executor`) to indicate which group of users may supervise and actually carry out a task.

*Activities* may make use of `Tools`, while *invocations* refer to `Services`. From the workflow system's perspective, both tools and services are external software components, which are accessible through interface adaptors. These are represented in the language as `Interfaces` (Fig. 4(b)), which may be connected to artefacts to indicate data flow (`input/output`). At run-time, both tools and services shall be triggered by the process execution engine, with parameters referring to data repository automatically supplied, so that the user does not have to care about managing data and files.




(a) Steps and contracts, services, tools (b) Component-artefact integration

**Fig. 4.** Integration process metamodel: auxiliary elements

**Contracts.** In order to guarantee that the result of a step is acceptable and the process can continue, the definition of *contracts* [23] is a well known paradigm. The idea is to *guard* both the input and output of a step by specific constraints. Thus, a `contract` is composed of a `precondition` and a `postcondition`. A precondition defines constraints that needs to be fulfilled by the input of the step in order to allow its execution, while the postcondition guarantees that the process can continue only if its constraints are satisfied by the output. This general definition of a contract allows to use arbitrary formalism (e.g., OCL, JML, etc.) to capture the pre- and postconditions, but could require further refinement of the metamodel with the additional elements of the used formalism.

A detailed example of configuration generation using our graph pattern based contract definition in the context of avionics domain (as part of our case study) can be found in Sec. 4.2.

## 4 Case Study: The DIANA Toolchain

DIANA (***D**istributed, equipment-**I**ndependent environment for **A**dvanced avio**N**ics **A**pplications* [6]) is an aeronautical research and development project. It aims at the definition of an advanced avionics platform named AIDA (Architecture for Independent Distributed Avionics), supporting (i) execution of object-oriented applications over virtual machines [24], (ii) high-level publish subscribe based communication, and (iii) the applicability of model driven system development (MDSD) in the avionics development domain.

The DIANA project aims to create an MDSD based tool chain for the analysis and generation of ARINC653 [25] real-time operating system (RTOS) configuration files from high-level specifications. Transforming these high-level models into RTOS-specific configuration artefacts is a complex task, which needs to bridge a large abstraction gap by integrating various tools. Moreover, critical design decisions are also made at this stage. For this reason, the use of intermediate domain specific models is advantageous to subdivide the process into well-defined steps and precisely define the interactions and interfaces among the tools used.

In order to introduce the DIANA approach Section 4.1 focuses on the models and metamodels used through the mapping process, while Section 4.2 gives an overview on the actual steps of the workflow.

### 4.1 Models

In the DIANA project the aim of the high-level Platform Independent Model (PIM) is to capture the high-level architectural view of the system along with the definition of the underlying implementation platform, while the Platform Specific Model (PSM) focuses on the communication details and service descriptions.

**Platform Independent Models.** In order to support already existing modeling tools and languages (e.g., Matlab Simulink, SysML, etc.) we use a common architecture description language called *Platform Independent Architecture Description Language* (PIADL) for architectural details by extracting relevant information from supported common off-the-shelf models. As for capturing the underlying platform (in our case ARINC653) we use a *Platform Description* model (PD) capable of describing common resource elements.

- PIADL aims to provide a platform independent architectural-level description of event-based and time-triggered embedded systems using message and publish/subscribe based communication between jobs, having roots in the PIM metamodel of the DECOS research project [26].
- The *Platform Description* (model) describes the resource building blocks, which are available in an AIDA Module to assemble the overall resources of an AIDA component. This mainly includes ARINC653 based elements such as modules, partitions, communication channels, etc. A small part of the metamodel is detailed in Section 5.2.
- In the context of the DIANA project we support *Matlab Simulink* as a source COTS language. We supports only a fraction of the language that conforms with the expressiveness of our PIADL to describe the high-level architecture of the system.

**Platform Specific Models.** The platform specific models are encapsulated in the *AIDA Integrated System Model* that contains all relevant low-level details of the modelled system. Essentially based on ARINC653, the integrated model provides extensions and exclusions to support the publish/subscribe communication and service based invocations. Its main parts are the following:

- The *Interface Control Document* (ICD) is used to describe data structures and low-level data representation of AIDA systems, interfaces and services to ease integration of the described element with other parts of the system. It supports both high-level (logical) and low-level (decoding) descriptions and was designed to be compatible with the ARINC653 and ARINC825 data and application interface descriptions.

– The *AIDA System Architecture* model identifying and describing the relations among all elements related to the AIDA system. More precisely the model focuses on the (i) details of the proposed publish/subscribe based communication, (ii) the multi-static configuration of the AIDA middleware and (iii) the detailed inner description of the partitions allocated for the AIDA system.

In order to support traceability – an essential requirement of DO-178B [1] certification –, a trace element is saved in the *Trace* model for all model elements of the PSM created during the mapping process. Such an element saves all PIM model segments that were used for the creation of a PSM model element. Additionally, trace information is also serialized into separate XMI files for each generated configuration file. In the current implementation traceability is hand-coded separately into each step of the development workflow.

### 4.2 Overview of the DIANA System Modeling Process

An extract of the defined workflow for the *DIANA System modeling* process is depicted in Figure 5, using a graphical concrete syntax of the process metamodel presented in Figure 3 and Figure 4.

The process starts with the definition of a `complete PIADL model` as the task of the `System architect` (represented by a human symbol). It can be either manually

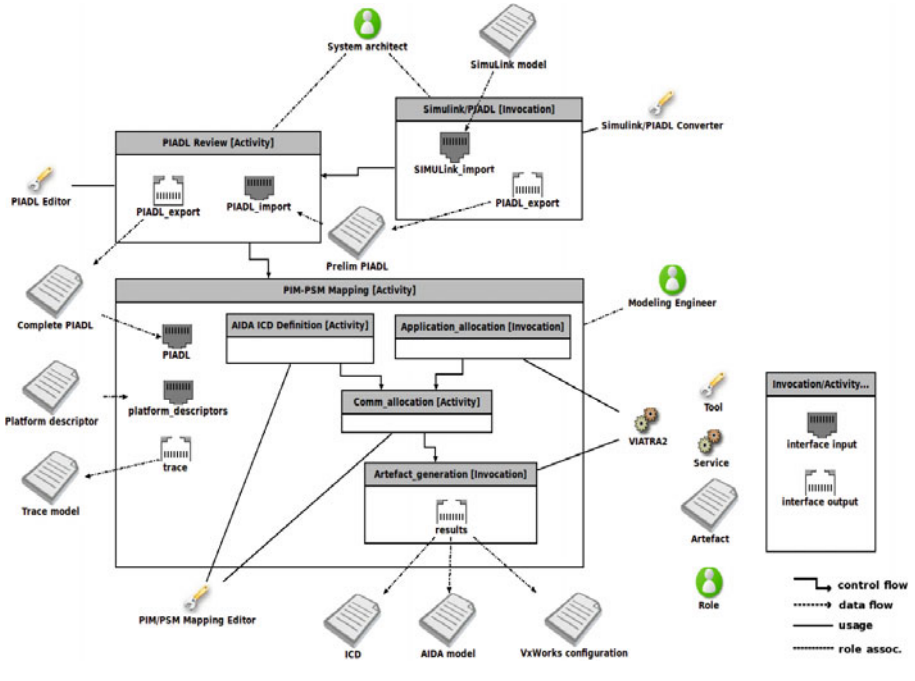



**Fig. 5.** Overview of the DIANA development process

defined using the (i) external PIADL editor (depicted by a wrench icon) as part of the PIADL review step or (ii) derived from a Simulink model.

The near one-to-one derivation is supported by the Simulink/PIADL Converter external tool used in the PIADL Review step. It has an input and an output interface figured by a grey and white socket symbol for the Simulink and the PIADL model, respectively. However, as some AIDA specific parameters cannot be directly derived it requires additional clarification from the system architect. For example, a *subsystem* block in the Simulink model is mapped to a *job* in the PIADL, but its modular redundancy value (how many instances of the job are required) is not present in the Simulink model.

The complete PIADL is then imported into the PIM/PSM mapping editor responsible for the analysis and definition of configuration tables interface descriptions. This work is done by the Modeling Engineer. Without going into more details it consists of 25 steps organized into the following main categories:

1. *Application allocation*: contains the PIM imports followed by the allocation of application instances to partitions and steps that define additional constraints on the allocation. It relies on the VIATRA2 framework and depicted by an invocation step.
2. *AIDA ICD definition*: steps related to the description of interfaces and services provided and required by applications. These are user driven mapping steps, where PIM types, messages, topics and services are refined with platform specific information like encoding, default value, etc. It is supported by the PIM/PSM mapping editor.
3. *Communication allocation*: involves steps in the PIM/PSM Mapping editor that carry out the allocation of inter-partition communication channels and the specification of ports residing on each end of these channels.
4. *Artefact generation*: contains steps that carry out the generation of AIDA middleware model, ARINC653 configuration files for the VxWorks real-time OS and the AIDA ICD descriptor.

Additionally, as a cross cutting aspect traceability information - depicted by the Trace model - is saved during the mapping process.

## 5 Graph Transformation in the DIANA Tool Chain

This section gives an overview how we successfully applied graph transformation based technology in various parts of the tool chain. Section 5.1 introduces a graph pattern based contract notation used to define conditions for steps, along with an example detailed in Section 5.2. Section 5.3 highlights how graph transformation is used for ARINC configuration generation.

### 5.1 Contracts as Graph Patterns

During a development process certain steps require external COTS tools (e.g., Matlab, SAL, etc.) or user interaction to perform their task. As mentioned in Section 3.1 we use

*contracts* to ensure that both the input and output of these steps satisfy their requirements. In our approach we used *graph patterns* to capture such contracts [27, 28] as we used on-the-fly evaluation based on incremental pattern matching [29]. However, it is important to note, that it would be possible to define these contracts by OCL and transform a large subset directly to the graph pattern formalism [30].

**Graph patterns** are frequently considered as the atomic units of model transformations [31]. They represent conditions (or constraints) that have to be satisfied by a part of the underlying instance model. In the following, we use the pattern language of the VIATRA2 framework [32]. In our interpretation a basic graph pattern consists of graph elements corresponding to the metamodel. As an addition for more complex pattern specification the language of VIATRA2 allows to define *alternate* (OR) *pattern bodies* for a pattern, with a meaning that the pattern is fulfilled if at least one of its bodies is fulfilled. A *negative application condition* (NAC) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [33].

**Contracts** are composed of a pre- and a postcondition. Both conditions are the conjunction of subconditions described by graph patterns, where a graph pattern is a disjunction of alternate pattern bodies [31]. A subcondition described by the graph pattern is either a *positive* or *negative* condition. A negative condition is satisfied by a model if it does not have a match in the underlying model. While a positive one is satisfied if its representing graph pattern has a match in the model. A further restriction on positive condition can be formulated by stating that they are satisfied iff their representing graph pattern has a *predefined* positive number (*Cardinality*) of matches.

### 5.2 Example

To demonstrate how contracts can be formulated using the defined approach consider the simplified *job allocation* step of the *Application Allocation* category (See in Section 4.2) using an external tool (the VIATRA2 framework). In this step the task is to allocate an IMA system defined by its jobs and partitions over a predefined cabinet structure and to minimize the number of *modules* used. An integrated modular avionics (IMA) system is composed of *Jobs* (also referred as applications), *Partitions*, *Modules* and *Cabinets*. *Jobs* are the atomic software blocks of the system defined by their memory requirement. Based on their criticality level, jobs are separated into two sets: *critical* and *simple* (non-critical). For critical jobs, double or triple modular redundancy is applied while for simple ones only one instance is allowed. *Partitions* are complex software components composed of jobs with a predefined free memory space. Jobs can be allocated to the partition as long as they fit into its memory space. *Modules* are SW components capable of hosting partitions. Finally, *Cabinets* are HW storages for maximum (in our example) two modules used to physically distribute elements of the system. Additionally a certain number of safety related requirements will also have to be satisfied: (i) a partition can only host jobs of one criticality level and (ii) instances of a

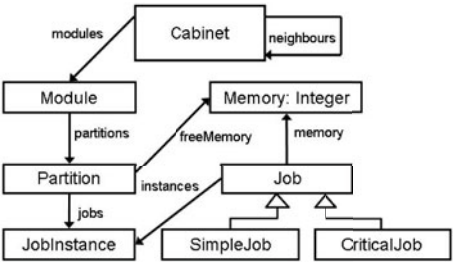



**Fig. 6.** Metamodel of an IMA architecture

certain critical job can not be allocated to the same partition. An excerpt of the Platform Description metamodel describing the detailed IMA system is depicted in Figure 6.

Based on this metamodel we defined the pre- and the postcondition of this step as depicted in Figure 7 and Figure 8, respectively. All subconditions in the pre- and the postcondition are defined as *positive* and *negative* conditions depicted with with + and - markings, respectively.

**Precondition.** For the definition of the precondition we rely only on that the model has at least one cabinet, one partition with its free memory defined and one job with an instance. These simple requirements are captured by the *cabinet*, *partition* and *job* graph patterns depicted in Figure 7.

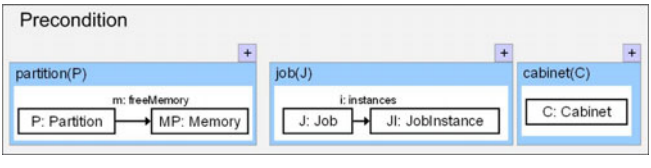



**Fig. 7.** Precondition of the DIANA application allocation step

**Postcondition.** The *jobInstancewithoutPartition*, *partitionwithoutModule* and *modulewithoutCabinet* subconditions describe that in a solution model each *JobInstance*, *Partition* and *Module* is allocated to a corresponding *Partition*, *Module* and *Cabinet*, respectively. For example, the *jobInstancewithoutPartition* subgoal captures its requirement using a double negation (NAC and negative constraint) stating that there is *no unallocated* job instance *JI* in the solution model. As the declarative graph pattern formalism has an implicit existential quantification, nested (double) negation is required to quantify elements universally. Similar double negation is used in case of the other two subgoals.

The rest formulates safety and memory requirements. The *partitionMemoryHigherThan0* pattern captures the simple memory constraint that all partitions must have higher than zero free memory. The safety requirement stating that a partition can only

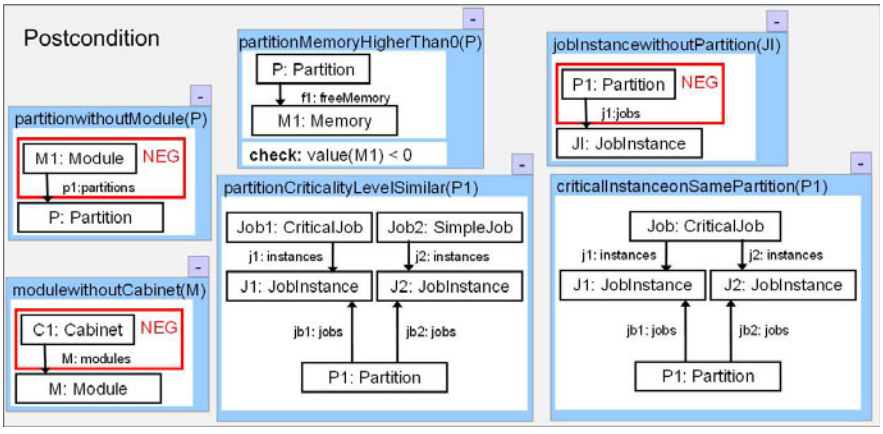



**Fig. 8.** Postcondition of the DIANA application allocation step

host jobs of one criticality level is captured by the *partitionCriticalityLevelSimilar* pattern. As it is a *negative constraint* it describes the (positive) case where the *P1* partition holds two job instances *J1* and *J2* of a simple and a critical job *Job1* and *Job2*, respectively. The *criticalInstanceonSamePartition* and *criticalInstanceonSameModule* patterns restrict in a similar way that no job instances *J1* and *J2* of a critical job *Job* can be allocated to the same partition *P1* or module *M1*.

Guarding the application allocation using this contract ensures that all applications are properly allocated and all safety requirements are fulfilled by the output model. As constraints are defined by graph patterns it gives rise to adapting constraint satisfaction programming techniques over graph transformation as in [34].

### 5.3 Configuration Generation by Graph Transformation

Model transformation based automatic code or configuration generation is one of the main driving forces [35,36] of model driven system development. It offers many advantages including the rapid development of high quality code, reduced number of errors injected during development and the consistency between the design and the code is retained, in comparison with a purely manual approach.

One aim of the DIANA project is to generate ARINC653 [25] XML based configuration files for VxWorks 653 RTOS from the created PSMs. A configuration file describes the internal structure of a module, namely: (i) allocated partitions and their memory layout, (ii) communication channels over sampling and queueing ports and (iv) health monitor tables for error detection and handling. During the PIM-PSM mapping process all relevant information required for the generation are specified and stored in the AIDA Integrated System model.

An example ARINC653 configuration snipplet is depicted in Figure 9. It captures the details of the *flight management non-system* partition, which has the highest *Level A* criticality as defined in [1], one queueing and four sampling ports and separate memory blocks for *code* and *data*. A port is defined with its *direction*, *maximum message size*

and *name*, where the sampling and the queuing ports have additionally *refresh rate* or *maximum number of messages* parameters, respectively. Finally, a memory block is defined by its *access* mode (e.g, read or write), *type* (code or data) and *size*.

```
- <Partition Criticality="LEVEL_A" EntryPoint="initial" PartitionIdentifier="p187" PartitionName="flight management" SystemPartition="false">
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="1088" Name="ACP_1_TO_PFC_1_AND_PFC_2_R1" RefreshRateSeconds="0.25" />
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="192" Name="PP_1_TO_SD_1___C0_R0" RefreshRateSeconds="0.1" />
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="576" Name="ZC_1_TO_PTC_1_AND_PTC_2_R0" RefreshRateSeconds="1.0" />
    <Sampling_Port Direction="DESTINATION" MaxMessageSize="576" Name="ZTP_1_TO_ZC_1_AND_SD_1_R1" RefreshRateSeconds="0.5" />
    <Queuing_Port Direction="DESTINATION" MaxMessageSize="64" Name="ACP_1_TO_PTC_1_AND_PTC_2_R0" MaxNbMessages="128" />
  </Partition>
+ <Partition Criticality="LEVEL_A" EntryPoint="initial" PartitionIdentifier="p939" PartitionName="IO processing" SystemPartition="true">
+ <Partition Criticality="LEVEL_A" EntryPoint="-" PartitionIdentifier="___mw821" PartitionName="Middleware_for_Channels" SystemPartition="true">
+ <Partition_Memory PartitionIdentifier="p996" PartitionName="flight controls">
- <Partition_Memory PartitionIdentifier="p187" PartitionName="flight management">
    <Memory_Requirements Access="READ_WRITE" SizeBytes="65536" Type="CODE" />
    <Memory_Requirements Access="READ_WRITE" SizeBytes="32768" Type="DATA" />
  </Partition_Memory>
```

**Fig. 9.** Example ARINC653 descriptor

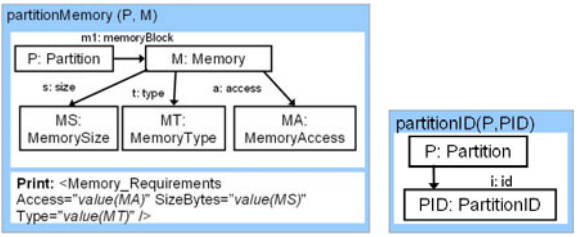
**Configuration Generation.** To generate the required XML format we based our code generator on the VIATRA2 framework, where configuration file templates are defined using graph transformation (GT) rules. This way GT rules define both the GT pattern that matches to the required elements in the model and generate code as a side effect. These GT rules do not modify the underlying model and thus their right and left hand sides are the same.

Without going into details, a part of the code generator responsible for the generation of the Partition_Memory XML subtree is depicted in Figure 10 and Listing 1.1.

The *partitionMemory* GT rule defines the template for the *Memory_Requirements* XML element. Its pattern matches to the partition *P* that has a memory *M* as its memory block. A memory has three attribute defined as memorySize *MS*, memoryType *MT* and memoryAccess *MA*. The *print* block defines the template that is printed out when the rule is applied. All three parameters value are retrieved using the *value* keyword.

The *partitionID* is an auxiliary pattern used to get the ID *PID* of the partition *P*.

To control the execution of the GT rules and define complex generators the VIATRA2 language [32] uses abstract state machines ASM [37]. ASMs provide complex model transformations with all the necessary control structures including the sequencing




(a) GT rule for memory block genera- (b) Partition with ID tion

**Fig. 10.** Example GT patterns and rules used for configuration generation

operator (*seq*), ASM rule invocation (*call*), variable declarations and updates (*let* and *update* constructs), *if-then-else* structures, non-deterministically selecting (*random*) constructs, iterative execution (applying a rule as long as possible (ALAP) *iterate*), the simultaneous rule application at all possible matches (locations) (*forall*) and single rule application on a single matching (*choose*).

The example code shown in Listing. 1.1 demonstrates how we defined our code generator using the *partitionMemory* rule and the *partitionID* pattern.

The outer *forall* rule is used to find all partitons *P* with their id *PID* in the model as defined by the *partitionID* pattern, and then execute its inner sequence on all matches separately. For each partition separate *Partition_Memory* XML elements are emitted out with their additional *PartitionIdentifier* and *PartitionName* parameters. As for the *Memory_Requirements* XML elements a *forall* rule invoking the *partitionMemory* GT rule is defined. The rule is invoked for all memory blocks *M* of partition *P*, where *P* is (at that point) already bound to a concrete partition by the outer forall.

The whole code generator is built up using similar snipplets.

```
... //memory block generation
forall P, PID with find partitonID(P, PID) do seq{
  println("<Partition_Memory PartitionIdentifier=\"+value(PID)
          +\" PartitionName=\"+name(P)+\">");
  forall M with apply partitionMemory(P,M); // GT rule as template
  println("</Partition_Memory>");
}
...
```

**Listing 1.1.** Partition_Memory code generator snipplet

## 6 Implementation of the Tool Integration Framework

To support the application of the high-level process modeling language presented in Sec. 3, we have created a prototype implementation for the tool integration framework. This framework provides the software infrastructure on which the case study of Sections 4 and 5 is executed. A main design goal was to integrate our solution to existing off-the-shelf tools that are used in industry practice; thus, both the process modeling infrastructure, as well as the execution environment rely on standard technologies as much as possible.

### 6.1 Execution Architecture

The execution of the process is facilitated by a service-oriented architecture, based on the jBoss jBPM [38] workflow execution engine and the Rational Jazz platform [39], as an integration middleware between tools, services, and the data repository. Building on this software environment, we have implemented a lightweight API that provides essential components, the overall architecture is shown in Fig. 11.

**Tool management.** A *Tool* or *Service* represents an external executable program that performs one or more tasks during the development. In order to be easily integrated, especially in the case of services, these software components should ideally be programmatically invocable, i.e., have the business functionality exposed to a well-defined

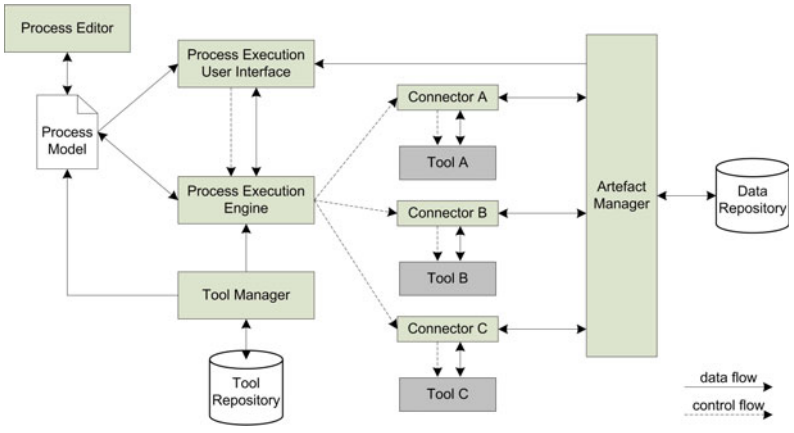



**Fig. 11.** Framework architecture

interface which is externally accessible (ranging from command line interfaces to library functions or even web services).

**Connectors.** Connectors are the components that provide uniform interface of the tools for the framework. The connector is also responsible for facilitating data flow between the tool and the artefact repository (optionally, support for explicit traceability may also be implemented in the Connector). The *Tool Manager* together with the *Tool Repository* serve as a service directory for available tools and services. It relies on the underlying service facilities of the Rational Jazz and OSGi platforms. These components are responsible for the lifecycle management (initialization, disposal) of integrated tools.

**Data management.** Models and artefacts of the development process are described (on a high abstraction level) in the Process Model. From this, a *storage metamodel* is generated, which contains dependency references between artefact classes, and includes metadata (such as creation timestamps, ownership and access control flags) as attributes. *Traceability* information is also handled as storage metamodel classes. The *Artefact Manager* is responsible for providing access through a service-oriented API (implemented as a tool/service interface) to data records stored in the *Data Repository* component.

**Process execution.** The executing processes can be managed and supervised using the *Process Execution User Interface*. In our prototypical implementation, it provides a control panel where (i) execution of tasks defined in the platform-specific process model can be initiated, (ii) the state of execution (i.e. the current process node, and the process variable values) can be observed, and (iii) versions of artefact instances and their related metadata can be managed.

The *Process Execution Engine* is responsible for the execution of the steps defined in the Process Model. The process model is mapped to a low-level executable language (jBoss jPDL [38]), which is executed in a customized jBPM instance. The jPDL description contains auxiliary information that is processed by *handler plugins*, so that

the process executor is able to invoke the integrated tools, services, and access the data repository.

### 6.2 Process Modeling Languages for Tool Integration

In practice, the domain-specific language of Sec. 3 is not the only means of designing a development process; in fact, several modeling languages may be involved on two levels of abstraction (Fig. 12).

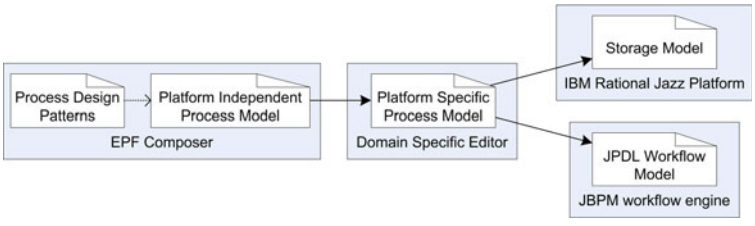



**Fig. 12.** Models and their relations used in process modeling and execution

**High-level process models.** In today's industrial practice, development processes are frequently captured in process description languages with a focus on methodology-compliance (i.e. enforcing design principles so that the actual development conforms to standard methods such as the Unified Process, or modern agile approaches such as XP or SCRUM). To address this need from a metamodeling perspective, the Software Process Engineering Metamodel (SPEM) [40] has been developed by the OMG. Since then, a number of SPEM-based tools have emerged, and IBM Rational's Method Composer is one of the most well-known of them. Along with its open-source version, the Eclipse Process Framework Composer [41] (shown in Fig. 13), they are based on pattern re-use by allowing to design according to process libraries that incorporate deep knowledge of both standard methodologies (e.g. OpenUP) and also organization-specific customizations.

| Presentation Name | Index | Predecessors |
|---|---|---|
| DIANA System Modeling Process | 0 | |
| Matlab Modeling | 1 | |
| PIM-PSM Modeling | 2 | |
| Simulink-PIADL Conversion | 3 | |
| PIADL Review | 4 | 3 |
| PIM-PSM Mapping | 5 | 4 |
| AIDA ICD Definition | 6 | |
| Application Allocation | 7 | |
| Communication Allocation | 8 | 6,7 |
| Artefact Generation | 9 | 8 |

**Fig. 13.** The DIANA process in the EPF Composer

As EPF's language includes support for the high level enumeration of roles and artefacts, with lightweight associations (such as responsibility, input-output), a "platform-independent" representation of development processes may be designed. Note that in

these models, all activities appear as *tasks* (Fig. 13), so there is no information present about which elements are used-guided and which are automated.

Thus, this high level model can mapped to our DSML by a VIATRA2 transformation, preserving the macro structure of the process, and importing an enumeration of roles, tools and artefacts. This domain-specific model has to be augmented manually to precisely specify how activities interact with artefacts, tools, services, and their interfaces.

**Storage models.** Two types of deployment models are generated from the platform-specific model: (i) the *Workflow Model* contains the description of the tool-chain to be executed in the format (augmented jPDL) that can be executed by the Process Execution Engine, and (ii) the *Storage Model*, which is the description of the data structure in a format that is needed to configure the Rational Jazz Data Repository.

In Jazz, *storage models* [42] are EMF/Ecore-compliant metamodels that define an object-oriented database schema in which artefacts can be stored. Inter-class references indicate cross-references between repository elements (for instance, such cross references may be used to determine which document instances need to be manipulated synchronously to maintain consistency).

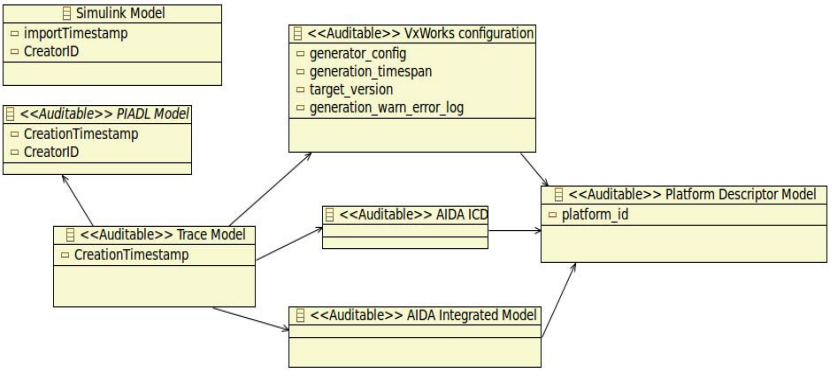



**Fig. 14.** Storage metamodel for the DIANA process

A sample storage model extract for the DIANA case study is shown in Fig. 14. Classes, tagged with the *Auditable* stereotype, are under versioning persistence management, and can store metadata as attributes. These attributes can be queried and processed without retrieving the artefacts in their entirety. Note that in this example, we do not record any traceability information between the Simulink model and the rest of artefacts, hence it is shown as a separate auditable entity in the storage metamodel.

Metadata attributes typically include lightweight traceability information (e.g. creation timestamps, creator IDs which may refer to a particular user or the ID of an automated service), and may also incorporate logs and traces (such as, for instance, the *generation_warn_error_log* attribute for the VxWorks configuration document, which contains the warning and error log emitted by the code generator during generation).

These auxiliary records, together with a complete *trace model* (also represented as a persistent and versioned artefact) play an important role in achieving *end-to-end traceability*.

Based on this storage model, a persistence plug-in is generated for the Jazz repository, which allows the tool connectors (as well as external components) to interact with the database on the artefact level. This interface currently only provides basic access functionality (queries and manipulations):

- getter functions for all document types and their metadata (e.g.`getVxWorks_configurations()`, `.getGenerator_config()` etc.), which directly retrieve data records from the artefact database and wrap them into EMF objects;
- getter functions for complex queries involving storage classes with cross references (e.g. `getTrace_model_transitive(String traceModelId)`, which fetches a particular trace model document together with all its referenced document instances);
- manipulation (setter) functions for all document types and their metadata attributes, which take EMF objects as data transfer parameters (e.g. `storeVxWorks_configuration(VxWorksConfigurationModel)`).

## 7 Related Work

The problem of tool integration has already been studied in many different research projects whose relationships to our proposed approach are now surveyed.

The UniForM WorkBench [43] can be considered as one of the earliest attempts for tool integration due to its built-in support for type safe communication between different tools, version and configuration management. Though system models can be interchanged in a type safe manner by the workbench, it cannot be considered as a model-based approach as a whole.

Several technology dependent approaches have already been proposed for tool integration purposes. One valuable representative of this group is R-OSGi [44], which supports the deployment of distributed applications on computers having the OSGi framework installed. Though the underlying OSGi framework has many advanced services, the centralized management (i.e., loading and unloading) of modules is an inconvenient property of R-OSGi. Another representative is the jETI system [45], which is a result of redesign and Java-based reimplementation of the Electronic Tool Integration platform, is an approach based on the Eclipse Plugin architecture whose technology dependency has been reduced by its Web Services support. The jABC submodule of the jETI system enhances Java development environments with remote component execution, high-level graphical coordination and dedicated control via formal methods.

The use of workflows for describing the tool integration process, which is a technique also employed in our approach, has been introduced in the bioinformatics domain in [46]. In this paper, the authors proposed to describe the cooperation of computational tools and data management modules by workflows.

The first form of metamodel-based tool integration appears in [22], which presents two orthogonal design patterns as well. The first pattern suggests the storage of metadata on a server, and the development of a model bus, on which tools can transfer models via a common model interface protocol. The other pattern proposes the use of workflows for describing the tool integration process in the ESML language.

*Model transformations in tool integration.* In the followings, tool integration solutions with model transformation support are presented.

In the authors' experience, VIATRA2, positioned as a dedicated model transformer, has been successfully applied both in scenarios where the abstraction gap (between source and target languages) was relatively small (such as code generation from MDA-style platform-specific models [19,47,48,49], or abstract-concrete syntax synchronization in domain-specific languages [50]), as well as mappings with strong abstractions (e.g., the generation of mathematical analysis models from design artefacts, for formal analysis purposes).

The IPSEN approach [51] outlined probably the first integration related scenario, where model transformation techniques played a key role. The aim of IPSEN was to construct an integrated software development environment (SDE) tool, which helped capturing both context-free (i.e., syntactic) and context-sensitive (i.e., graph-based) aspects of languages by textual and graphical editors, respectively. The technique of graph transformation has been heavily used for the development of the tool especially for specifying constraints and translations in the context-sensitive domain.

ModelCVS [52] employs (i) semantic technologies in forms of ontologies to partly automate the integration process, and (ii) QVT transformations, which are generated from these ontology descriptions. As distinctive features, ModelCVS uses Subversion for versioning, EMF and MOF-based metamodels for model representation, and a generic workflow ontology for defining processes. In contrast to our approach, ModelCVS prepares adapters for tools and not for models as these latters are stored in a central repository. Additionally, model transformations are used in ModelCVS for the synchronization of models, and not for the definition of the integration process.

From the model transformation point of view, a similar setup can be found in MOFLON [53, 54]. Transformations are again used for model synchronization, but in this case, they are defined by triple graph grammars. MOFLON operates on JMI and MOF 2.0 based models.

TopCased ("The Open source toolkit for Critical Systems") [55] is a software environment primarily dedicated to the realization of critical embedded systems including hardware and/or software. Topcased promotes model-driven engineering and formal methods as key technologies, such as a model bus-based architecture supporting standard modeling technologies such as EMF, AADL, UML-MARTE, and SysML. For model transformations, TopCased uses ATL [56].

The recent EU projects of ModelWare [57] and MODELPLEX [58] outline techniques that show certain similarity to our approach. ModelWare aimed at defining and developing the complete infrastructure required for large-scale deployment of MDD strategies and validating it in several business domains. It can (i) provide transparent integration across model, tool, platform, machine boundaries; (ii) support the creation of distributed, multi-user tool chains; (iii) handle many metamodels and artefacts; (iv) integrate interactive and non-interactive tools; and (v) use different technologies for communication. ModelWare offers a process modeling framework, and a model bus for exchanging high-level data that are either Java-based or described by Web Services. On the other hand, it lacks model transformation support, which has only been added in its successor MODELPLEX project. MODELPLEX has a SPEM2 based toolset for

supporting the enactment and execution of processes and is integratable with workflow and project management tools as well.

The clear separation of PIMs and PSMs, which specify tool integration processes with different levels of details can only be found in research projects GENESYS [9] and DECOS [5], which propose a cross-domain architecture for embedded systems, and a model-driven development process for avionics systems, respectively. As distinctive features, GENESYS supports (i) different modeling languages including UML and many of its profiles, (ii) a service-oriented development of subsystems, (iii) both uni- and bidirectional model transformations with manual, semi-automatic, automatic execution.

## 8 Conclusion

In the paper, we proposed a tool integration framework, which is centered around a high-level domain-specific language for process modeling to closely align tool integration and development scenarios. With this approach, model-driven development processes can be described precisely, in detail that is sufficient to capture what can and should be automated, but also flexible enough to support user-guided steps as well.

Based on our experience in designing and implementing tool chains primarily in the embedded and service-oriented domains, a key issue is to precisely specify and validate the individual elementary steps of the development tool chain. For this purpose, we adapted graph patterns to formally specify contracts for each step. Furthermore, model transformations provided by graph transformation techniques were responsible for fully automating certain steps in the tool chain (like code generation or model analysis tasks).

In addition to the process-driven specification of tool integration chains, we have also presented an execution framework, which is rooted on various research projects at the group. This framework is built to accommodate a wide spectrum of Eclipse-based or external) tools, and automatically execute development processes designed with our modeling language.

As future work, we primarily aim at including support for advanced model bus services, such as versioning, model merge, and automatic traceability information generation. Additionally, we are planning to integrate advanced support for automated traceability based on change-driven transformations introduced in [50].

## References

1. RTCA - Radio Technical Commission for Aeronautic: Software Considerations in Airborne Systems and Equipment Certification, DO-178B (1992), https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=633
2. Rushby, J.: Runtime Certification. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 21–35. Springer, Heidelberg (2008)
3. Kornecki, A.J., Zalewski, J.: The Qualification of Software Development Tools from the DO-178B Perspective. Journal of Defense Software Engineering (April 2006), http://www.stsc.hill.af.mil/crosstalk/2006/04/0604KorneckiZalewski.html
4. Miller, S.P.: Certification Issues in Model Based Development Rockwell Collins

5. The DECOS Project: DECOS - Dependable Embedded Components and Systems, http://www.decos.at/
6. The DIANA Project Consortium: DIANA (Distributed, equipment Independent environment for Advanced avioNic Application) EU FP6 Research Project, http://dianaproject.com
7. The SENSORIA Project: The SENSORIA website, http://www.sensoria-ist.eu
8. The MOGENTES Project : MOGENTES (Model-based Generation of Tests for Dependable Embedded Systems) EU FP7 Research Project, http://mogentes.eu
9. The GENESYS Project: GENESYS - GENeric Embedded SYStem, http://www.genesys-platform.eu/
10. The INDEXYS Project: INDEXYS - INDustrial EXploitation of the genesYS cross-domain architecture, http://www.indexys.eu/
11. Ráth, I., Vágó, D., Varró, D.: Design-time Simulation of Domain-specific Models By Incremental Pattern Matching. In: 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2008)
12. Pintér, G., Majzik, I.: Runtime Verification of Statechart Implementations. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 148–172. Springer, Heidelberg (2005)
13. Sisak, Á., Pintér, G., Majzik, I.: Automated Verification of Complex Behavioral Models Using the SAL Model Checker. In: Tarnai, G., Schnieder, E. (eds.) Formal Methods for Automation and Safety in Railway and Automotive Systems (Proceedings of the FORMS-2008 Conference), Budapest, Hungary, LH́armattan (2008)
14. Partaricza, A.: Systematic generation of dependability cases from functional models. In: Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMAT 2008), Budapest, Hungary (2007)
15. Majzik, I., Pataricza, A., Bondavalli, A.: Stochastic dependability analysis of system architecture based on uml models. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677, pp. 219–244. Springer, Heidelberg (2003)
16. Schoitsch, E., Althammer, E., Eriksson, H., Vinter, J., Gönczy, L., Pataricza, A., Csertán, G.: Validation and Certification of Safety-Critical Embedded Systems - the DECOS Test Bench. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 372–385. Springer, Heidelberg (2006)
17. Balogh, A., Pataricza, A., Ráth, I.: Automated verification and validation of domain specific languages and their applications. In: Proceedings of the 4th World Congress for Software Quality, Bethesda, USA, pp. 1–6 (2009)
18. Pintér, G., Majzik, I.: Model Based Automatic Code Generation for Embedded Systems. In: Proceedings of the Regional Conference on Embedded and Ambient Systems (RCEAS 2007), Budapest, Hungary, pp. 97–106 (2007)
19. Gönczy, L., Ávéd, J., Varró, D.: Model-based Deployment of Web Services to Standards-compliant Middleware. In: Isaias, P., Miguel Baptista Nunes, I.J.M. (eds.) Proc. of the Iadis International Conference on WWW/Internet 2006 (ICWI 2006), Iadis Press (2006)
20. Kövi, A., Varró, D.: An eclipse-based framework for ais service configurations. In: Malek, M., Reitenspieß, M., van Moorsel, A. (eds.) ISAS 2007. LNCS, vol. 4526, pp. 110–126. Springer, Heidelberg (2007)
21. Pintér, G., Majzik, I.: Automatic Generation of Executable Assertions for Runtime Checking Temporal Requirements. In: Dal Cin, M., Bondavalli, A., Suri, N. (eds.) Proceedings of the 9th IEEE International Symposium on High Assurance Systems Engineering (HASE 2005), Heidelberg, Germany, October 12-14, pp. 111–120 (2005)
22. Karsai, G., Lang, A., Neema, S.: Design Patterns for Open Tool Integration. Software and Systems Modeling 4(2), 157–170 (2004)
23. Meyer, B.: Applying ”design by contract”. IEEE Computer 25(10), 40–51 (1992)

24. Locke, C.D.: Safety critical java™technology. In: JTRES 2006: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems, pp. 95–96. ACM, New York (2006)
25. ARINC - Aeronautical Radio, Incorporated: A653 - Avionics Application Software Standard Interface, https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=633
26. DECOS - Dependabe Embedded Components and Systems consortium : The DECOS Platform Independent Metamodel, public deliverable, http://www.inf.mit.bme.hu/decoscd/deliverables/DECOS_deliv_PIM_Metamodel.pdf
27. Baar, T.: OCL and graph-transformations - A Symbiotic Alliance to Alleviate the Frame Problem. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 20–31. Springer, Heidelberg (2006)
28. Azab, K., Habel, A.: High-level programs and program conditions. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 211–225. Springer, Heidelberg (2008)
29. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT 2008, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (2008)
30. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. Electron. Notes Theor. Comput. Sci. 211, 159–170 (2008)
31. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 214–234 (2007)
32. Balogh, A., Varró, D.: Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In: ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006) (2006) (in press)
33. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004)
34. Horváth, Á., Varró, D.: CSP(M): Constraint Satisfaction Programming over Models. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 107–121. Springer, Heidelberg (2009)
35. Hemel, Z., Kats, L.C.L., Visser, E.: Code generation by model transformation. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 183–198. Springer, Heidelberg (2008)
36. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Towards graph transformation based generation of visual editors using eclipse. Electr. Notes Theor. Comput. Sci. 127(4), 127–143 (2005)
37. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
38. Koenig, J.: JBoss jBPM White Paper. Technical report, The JBoss Group / Riseforth.com (2004), http://jbossgroup.com/pdf/jbpm_whitepaper.pdf
39. IBM Rational: Jazz Community Site, http://jazz.net/
40. The Object Management Group: Software Process Engineering Metamodel, version 2.0 (2008), http://www.omg.org/technology/documents/formal/spem.htm
41. The EPF Project: The Eclipse Process Framework website, http://www.eclipse.org/epf/
42. Haumer, P.: Increasing Development Knowledge with Eclipse Process Framework Composer. Eclipse Review (2006), http://haumer.net/rational/publications.html
43. Einar W. Karlsen: The UniForM WorkBench: A Higher Order Tool Integration Framework. Lecture Notes in Computer Science **1641** (1999) 266–280

44. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 1–20. Springer, Heidelberg (2007)
45. Margaria, T., Nagel, R., Steffen, B.: jETI: A Tool for Remote Tool Integration. LNCS, vol. 2440, pp. 557–562. Springer, Heidelberg (2005)
46. Corradini, F., Mariani, L., Merelli, E.: An Agent-based Approach for Tool Integration. International Journal on Software Tools for Technology Transfer 6(3), 231–244 (2004)
47. Gönczy, L., Déri, Z., Varró, D.: Model Driven Performability Analysis of Service Configurations with Reliable Messaging. In: Proc. of Model Driven Web Engineering Workshop (MDWE 2008) (2008)
48. Gönczy, L., Déri, Z., Varró, D.a.: Model transformations for performability analysis of service configurations, pp. 153–166 (2009)
49. Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 342–356. Springer, Heidelberg (2009)
50. Ráth, I., Ökrös, A., Varró, D.: Synchronization of abstract and concrete syntax in domain-specific modeling languages. Journal of Software and Systems Modeling (2009) (accepted)
51. Klein, P., Nagl, M., Schürr, A.: IPSEN Tools. In: [59], pp. 215–266. World Scientific, Singapore (1999)
52. Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Wimmer, M.: On Models and Ontologies – A Layered Approach for Model-based Tool Integration. In: Proceedings of the Modellierung 2006, pp. 11–27 (2006)
53. Klar, F., Rose, S., Schürr, A.: A Meta-model Driven Tool Integration Development Process. Lecture Notes in Business Information Processing, vol. 5, pp. 201–212 (2008)
54. Amelunxen, C., Klar, F., Königs, A., Rötschke, T., Schürr, A.: Metamodel-based Tool Integration with MOFLON. In: International Conference on Software Engineering, pp. 807–810. ACM, New York (2008)
55. The TOPCASED Project: TOPCASED - The Open-Source Toolkit for Critical Systems, http://www.topcased.org/
56. Canalsm, A., Le Camus, C., Feau, M., et al.: An Operational Use of ATL: Integration of Model and Meta Model Transformations in the TOPCASED Project. In: Ouwehand, L. (ed.) Proc. of the DASIA 2006 - Data Systems in Aerospace Conference, European Space Agency, p. 40 (2006), http://adsabs.harvard.edu/abs/2006ESASP.630E.40C
57. The ModelWare Project: ModelWare - MODELling solution for softWARE systems, http://www.modelware-ist.org/
58. The MODELPLEX Project: MODELPLEX - Modeling Solution for Complex Systems, http://www.modelplex-ist.org/
59. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook on Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)