The Formalism Transformation Graph as a Guide to Model Driven Engineering

Levi Lúcio[†], Sadaf Mustafiz[†], Joachim Denil[‡],
Bart Meyers[‡], Hans Vangheluwe[‡†]

[†]School of Computer Science, McGill University, Canada [**]
[‡]University of Antwerp, Belgium
`{levi,sadaf,hv}@cs.mcgill.ca`
`{Joachim.Denil,Bart.Meyers,Hans.Vangheluwe}@ua.ac.be`

Abstract. In recent years, many new concepts, methodologies, and tools have emerged, which have made Model Driven Engineering (MDE) more usable, precise and automated. A MDE process is very often dependent on the domain. Thus, means for composing and customizing MDE activities are increasingly necessary. In this paper, we propose the FTG+PM framework that acts as a guide for carrying out model transformations, and as a basis for unifying key MDE practices, namely multi-paradigm modelling, meta-modelling, and model-transformation. The FTG+PM consists of the Formalism Transformation Graph (FTG) and its complement, the Process Model (PM), and charts all kinds of activities in the MDE lifecycle such as requirements development, domain-specific design, verification, simulation, analysis, calibration, deployment, code generation, execution, etc. The FTG describes in an explicit and precise way, formalisms, and their relationships as transformations between formalisms. The PM defines an actual MDE process using these formalisms and transformations. We illustrate the proposed FTG+PM approach through the design of an automated power window, a case study from the automotive domain.

1 Introduction

In recent times, model driven engineering (MDE) has been adopted in industrial projects in widely varying domains. The promises of MDE regarding traditional software development methods are many. The most important ones are: better management of the complexity of software development by making use of powerful abstractions; better management of the requirements for the system coming from the stakeholders, by both exposing the logic of the system in languages that are understandable by non programmers as well as fast re-generation of code by using automated model transformations; less bugs in the final software product given that automation helps eliminate errors and usage of formal verification tools raises confidence of correctness; and finally automated documentation generation from domain specific models. If achieved, all these benefits would

---

[**] Part of this work has been developed in the context of the NECSIS project, funded by the Automotive Partnership Canada

translate in potentially faster, cheaper and more reliable software development techniques than the ones traditionally used.

Several important concepts and associated fields of study have emerged or have been adopted and further developed by the efforts of the MDE community. *Model transformations*, *domain specific modelling*, *requirements engineering*, *verification and validation*, *multi-paradigm modelling*, *model composition*, *simulation*, *calibration*, *deployment*, *code generation*, etc. are often proposed in the form of tools, methodologies or frameworks to help alleviate issues in the application of MDE. However, to the best of our knowledge, the challenges and benefits arising from the conjugation and synergies of all these concepts during the application of MDE are yet to be explored. This is partially due to the fact that most of the tools, methodologies or frameworks proposed by the community often focus in-depth on technically challenging issues, while the broader picture of the systematic integration of those technical and methodological solutions remains, for the time being, to be explored. An additional difficulty often faced by MDE researchers is the limited access to the software development tools, methodologies and models used in real industrial settings. This is often due to the fact that companies that do apply MDE techniques during software development do not want to expose their development processes or data either by fear of loss of competitive edge, or simply by lack of time and resources to share their know-how.

The goal of our work is to provide a complete and detailed process architecture for model-driven software development by unifying key MDE practices. We propose FTG+PM framework intended to guide developers throughout the MDE lifecycle. The FTG+PM is comprised of the Formalism Transformation Graph (FTG) and its complement, the Process Model (PM). The idea behind the FTG is similar to the Formalism Transformation Lattice for coupling different formalisms as proposed by Vangheluwe et al in [45]. We go a step beyond multi-formalism modelling, and use the notion of multi-paradigm modelling [31] as the basis of our work. Model transformation is a key element in our FTG+PM. Our FTG+PM addresses the need for domain-specific modelling, and an instance of the FTG includes domain-specific formalisms and transformations between them that allow capturing a map of the process used to develop software within a given domain. The PM introduced as part of the FTG+PM can be used to precisely model the control flow between the transformation activities taking place throughout the software development lifecycle starting from requirements analysis and design to verification, simulation, and deployment.

We have worked with automotive systems as our target domain, but we believe that the FTG+PM can be applied in general in a broad range of domains. In particular, we demonstrate the capabilities of the FTG+PM through the design of an automated power window. The case study is of inherent complexity, non-trivial in nature, and representative of industrial case studies. The formalisms used in the FTG are appropriate to the levels of abstraction used at different stages of the modelling process. Discrete-time, continuous-time, discrete-event, and hybrid formalisms are included. The MDE process is entirely based on models and transformations, starting from domain specific requirements and design models aimed at describing control systems and their environment and finishing with Automotive Open System Architecture (AUTOSAR) [4] code.

This paper is organised as follows: Section 2 provides background information on meta-modelling, model transformation, and multi-paradigm modelling. Section 3 describes the FT&P and illustrates it using the power window case study. Section 4 gives a formal definition of the formalism transformation graph (FTG) and the process model (PM). Section 5 discusses our contibutions and possible improvements of FTG+PM. Section 6 presents related work in this area and compares our contribution to it and Section 7 draws some conclusions.

2 Background

Model Driven Engineering (MDE) encompasses both a set of tools and a methodological approach to the development of software. MDE advocates building and using abstractions of processes (and associated artefacts) the software engineer is trying to automate, thus making them easier to understand, verify, and simulate than computer programs.

Within the context of this paper, we have chosen to follow the terminology as presented in [17]). A *model* is completely described by its abstract syntax (its structure), concrete syntax (its visualisation) and semantics (its unique and precise meaning). A *language* is a possibly infinite set of (abstract syntax) models. This set can be concisely described by means of e.g., a grammar or a metamodel. No semantics or concrete syntax is given to these models.

Domain Specific Modelling (DSM) captures the fact that certain languages or classes of languages, called Domain Specific Languages (DSLs), are appropriate to describe models in certain domains. A white paper on the subject from Metacase^{TM} [27] presents anecdotal evidence that DSLs can boost productivity by a factor of 10, based on experiences with developing operating systems for cell phones for Nokia^{TM} and Lucent^{TM}. DSM has led to the development of formalisms and tools such as EMF and GMF [30], AToM^3 [11] or Microsoft's DSL Tools^{TM} [9].

Model transformations are the heart and soul of model-driven software development, as stated by Sendall and Kozaczynski [40]. Model transformation involves mapping of source models in one or more formalisms to target models in one or more formalisms using a set of transformation rules. Having an automated process for creating and modifying models leads to reduced effort and errors on the software engineer's part.

Implementations for transformation languages such as ATL [2] or QVT [13], and for graph transformations (as used in AToM^3) have been developed in the last few years and provide stable platforms for describing and executing model transformations.

Multi-Paradigm Modelling (MPM), as introduced by Mosterman and Vangheluwe in [31], is a perspective on systems development that advocates not only that models should be built at the right *levels of abstraction* regarding their purpose, using the most appropriate *formalisms*, but also that automatic *model transformations* should be used to pass information from one representation to another during development. In this case, it is thus desirable to consider modelling as an activity that spans different paradigms.

The main advantage that is claimed of such an approach is that the software engineer can benefit from the already existing multitude of languages and associated tools for describing and automating software development activities – while pushing the task of transforming data in between formalisms to (semi-)automated transformations. Another possible advantage of MPM is the fact that toolsets for implementing a particular software development methodology become flexible. This is thanks to the fact that formalisms and transformations may be potentially plugged in and out of a development toolset given their explicit representation.

3 FTG+PM: The Power Window Case Study

The goal of this section is to introduce the FTG+PM framework. The language used to define FTG+PM consists of two sub languages: the Formalism Transformation Graph (FTG) language, which allows declaring a set of languages available to model within a given domain as well as available transformations between those languages; and a Process Model (PM) language, which is used to describe the control and data flow between MDE activities. We illustrate our work using the power window case study from the automotive domain.

A power window is basically an electrically powered window. Such devices exist in the majority of the automobiles produced today. The basic controls of a power window include lifting and descending the window, but an increasing set of functionalities is being added to improve the comfort and security of the vehicle's passengers. To manage this complexity while reducing costs, automotive manufacturers use software to handle the operation and overall control of such devices. However, because of the fact that a power window is a physical device that may come into direct contact with humans, it becomes imperative that sound construction and verification methodologies are used to build such software.

In Figure 1 we depict a condensed version of the FTG+PM we have built for developing Power Window software. The FTG is shown on the left side, the PM is shown on the right side. The power window FTG+PM was built based on experiments we have performed while developing software development processes for the automotive industry. Notice that in the FTG (left side of the FTG+PM of Figure 1) a set of domain specific formalisms are defined as labelled rectangles. Transformations between those formalisms are depicted as labelled small circles. On the PM (right side of the FTG+PM of Figure 1) a diagram with a set of ordered tasks necessary to produce the power window control code is laid out. The language used for the PM is the UML Activity Diagram 2.0 language [35]. The labelled round edged rectangles (actions) in the Activity Diagram correspond to executions of the transformations declared on the power window FTG. Labelled

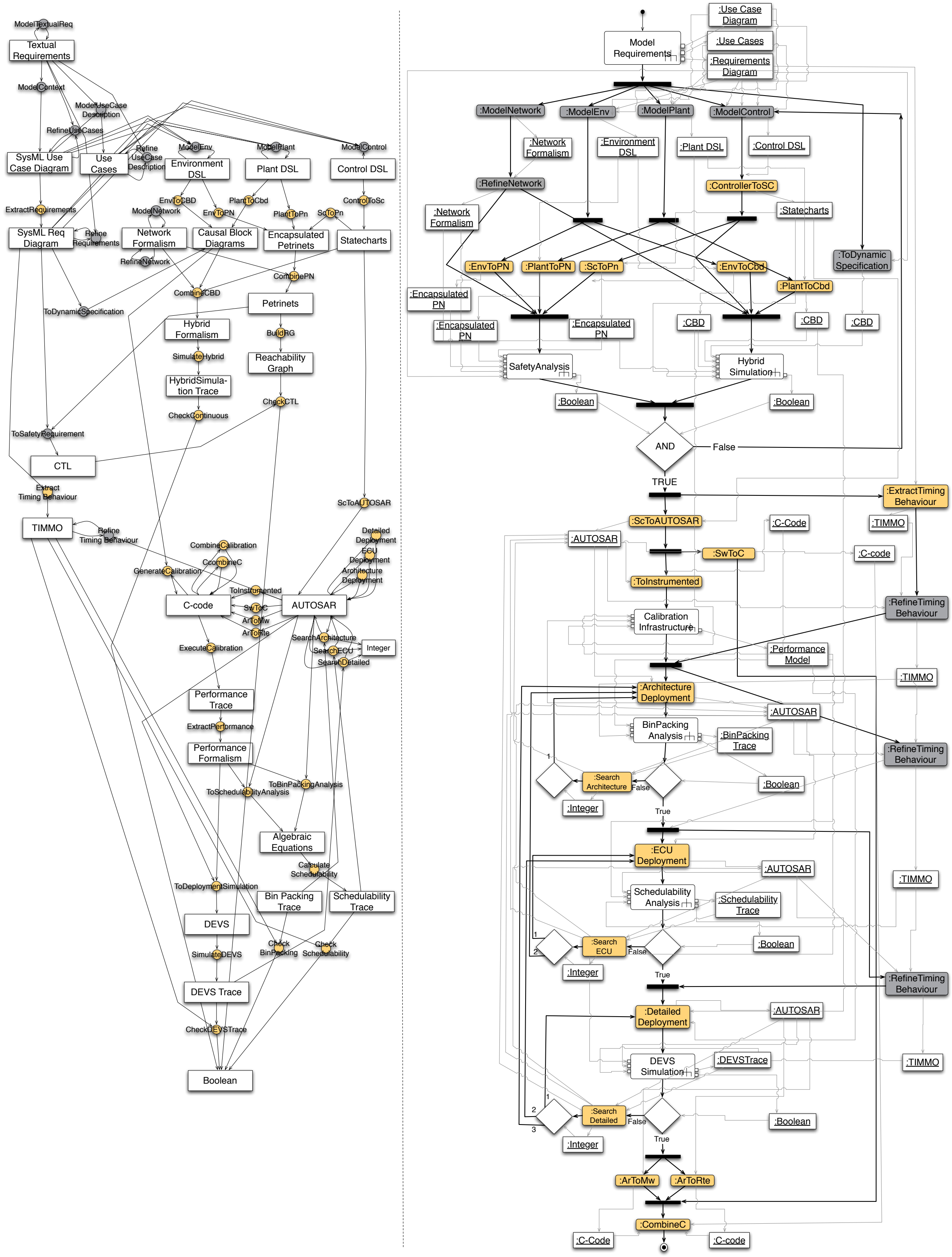

Fig. 1. Power Window: FTG (left) and PM (right)

square edged rectangles (data objects) in the PM correspond to models that are consumed or produced by actions. A model is an instance of the formalisms declared on the power window FTG with the same label. Notice that on the PM side the thin arrows indicate data flow, while thick arrows indicate control flow. Similar to the models, the arrows must also have corresponding arrow in the FTG, meaning that their input and output nodes must correspond. Similar to Activity Diagrams we also use control flow constructs for a PM like joins and forks, represented as horizontal bars, and decisions, represented by diamonds. The formalised meaning of the FTG+PM will be presented in depth in Section 4.

The power window FTG+PM of Figure 1 contains several phases, that are sometimes executed in parallel. These contain (1) Requirements Engineering, (2) Design, (3) Verification, (4) Simulation, (5) Calibration, (6) Deployment and finally (7) Code Generation, which are described below. Due to this paper's space constraints we provide detailed descriptions of only verification and deployment. However, most of the languages defined within the FTG+PM for the power window, with the exception of requirements, has been described in [24].

3.1 Requirements Engineering

Before any design activities can start, the requirements need to be formalised so they can be used by the engineers. Starting from the *textual description* containing the features and constraints of the power window, a context diagram is modelled using the *SysML use case diagram*. The use cases are further refined and complimented with the *use case descriptions*. Finally, the requirements are captured more formally with a *SysML requirements diagram*. Note that these transformations are usually done manually by the requirements engineers though some automatic transformations can be used to populate the use case diagram and requirements diagram. The manual transformations are shown greyed out in the FTG.

3.2 Design

When given the task to build the control system for a power window, engineers will take two variables into consideration: (1) the physical power window itself, which is composed of the glass window, the mechanical lift, the electrical engine and some sensors for detecting for example window position or window collision events; (2) the environment with which the system (controller plus power window) interacts, which will include both human actors as well as other subsystems of the vehicle – e.g. the central locking system or the ignition system. This idea is the same as followed by Mosterman and Vangheluwe in [31]. According to control theory [12], the control software system acts as the *controller*, the physical power window with all its mechanical and electrical components as the *process* (also called the *plant*), and the human actors and other vehicle subsystems as the *environment*.

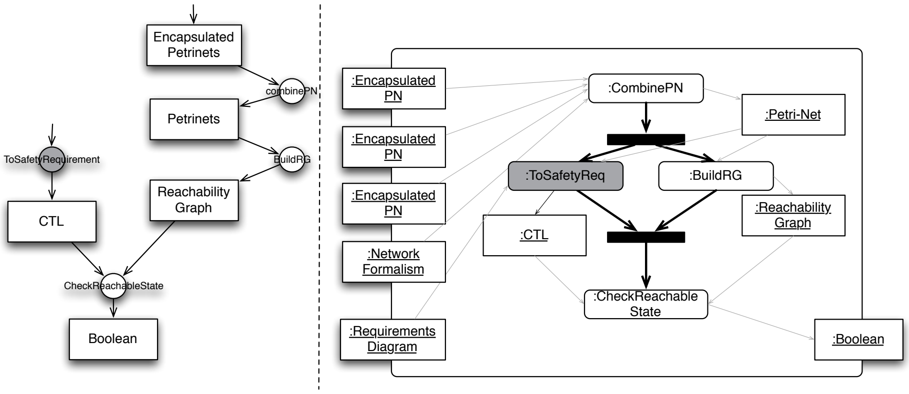

Fig. 2. Control Design FTG+PM Slice, with FTG on the left and PM on the right

Using the requirements, engineers start the design activities using domain specific languages (DSL) for the *Environment*, *Plant*, and *Controller*. The *Network* is used to combine the three design languages and identify the interfaces between them.

3.3 Verification

To assure that there are no safety issues with the modelled control logic, a formal verification can be done. The domain specific models used for defining the plant, environment and the control logic are transformed to Petri nets [36] where reachability properties are checked. Of course it is also necessary to evolve requirements to a language that can be used to check the Petri nets.

In Figure 2, we present the full safety analysis part of the power window PM, along with the corresponding subset of the FTG. Notice that the safety analysis block is displayed in its collapsed form in the complete FTG+PM in Figure 1.

Figure 2 shows a part of the FTG+PM. It uses five models that are the result of previous activities (shown on the left of the PM). We see that the *combinePN* activity takes as inputs three encapsulated Petri nets[1] derived from the environment, plant and control domain specific models in Figure 1, as well as a network model that specifies how those three models communicate. As data output, the *CombinePN* activity produces a *Place/Transition Petri net* (non-modular), which is the result of the fusion of the three input modular Petri nets according to the input *Network* model.

Following the *CombinePN* activity, the *ToSafetyReq* and *BuildRG* activities should be executed in parallel. The *ToSafetyReq* activity is greyed out since it needs human intervention. It takes as inputs a model of the safety requirements for the power window, as well as the combined Petri net model including the behaviour of the whole system, and outputs a set of CTL (Computation Tree

---

[1] *encapsulated Petri nets* are a modular Petri net formalism, where transitions can be connected to an encapsulating module's ports. Module's ports can then be connected by a *Network* formalism.

Logic) formulas encoding the requirements. On the other hand the *BuildRG* action is automatic and allows building the reachability graph for the combined Petri net model. The join bar enforces that both the CTL formulas and the reachability graph are produced before the *CheckReachableState* action is executed. This last action verifies if the reachability graph adheres to the formulas built from the requirements and produces a boolean as output.

By using FTG+PM the causal relations between the different activities emerges explicitly.

3.4 Simulation

On the other hand, the continuous behaviour of the up-and-downward movement of the window is simulated using a hybrid formalism. The hybrid simulation contains the environment and plant models transformed into Causal Block Diagrams[2] (CBD) and the controller in the Statecharts formalism. The process of verifying the continuous behaviour is very similar to the Safety Analysis, presented in section 3.3 though as a requirements language CBDs are also used.

3.5 Deployment

After the software has been created and verified, the software has to be deployed onto a hardware architecture. This hardware architecture contains a set of electronic control units (ECU) that are connected using a network. Each ECU can execute a set of related and unrelated software components. To allow this, AUTOSAR defines a standardised middleware containing a real-time operating system, a communication stack and drivers to access the peripherals like analog-digital converters, timers and others. Software components can be distributed freely among the available hardware units. Other tasks need to be done like mapping the software functions to tasks, mapping signals to messages and choosing from a multitude of deployment options in the middleware. These choices give the engineer a lot of flexibility that can result in non-feasible solutions where the spatial and temporal requirements are violated. On the other hand it allows to search the deployment space for optimal solutions in terms of cost, energy consumption and other extra-functional properties.

In our power window case study, we take a platform-based design method[39] for exploring the deployment space with the goal of creating a feasible deployment solution in terms of real-time behaviour. Platform-based design introduces clear abstraction layers where certain properties can be checked. Real-time behaviour can be checked in three stages that step-wise prune the deployment space: (1) after mapping the software to the hardware using a simple bin packing check, (2) after mapping the software functions to tasks and messages to the bus using schedulability analysis and (3) after setting all the parameters in the middleware using a low-level deployment simulation.

---

[2] Causal Block Diagrams are a general-purpose formalism used for modelling causal, continuous-time systems, mainly used in tools like Simulink

Figure 3(a) shows the actions involved in checking a single solution at the level of schedulability analysis. *ToSchedulabilityAnalysis* takes a single AUTOSAR solution and a performance model as input to derive set of algebraic equations which are subsequently executed. This execution, modelled as *CalculateSchedulability*, produces a trace containing the worst-case execution times of the software functions. Afterwards the trace is compared to the requirements, expressed using the TIMMO-language [8] in *CheckSchedulabilityTrace*, which produces a boolean denoting whether the requirements are met. When the result is not satisfying the requirements, a backtracking step is taken so that new deployment solutions can be explored. The process continues until a feasible solution is found. Transforming to another language, executing this new model to obtain execution traces and comparing these traces to check a certain property is a common activity that can be seen as a pattern for all three deployment levels in the FTG+PM of Figure 1.

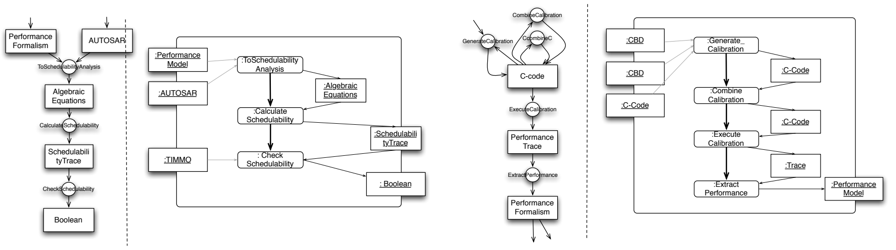

Fig. 3. (a): Schedulability Analysis Slice, and (b): Calibration slice

3.6 Calibration

In the previous paragraphs we assumed that a performance model was readily available to use during the deployment space exploration. To build the performance model, we can also use fully automated generative MDE techniques. This process is depicted in Figure 3(b), where the plant model, environment model and instrumented source code are combined and executed in a Hardware-in-the-loop environment[3] giving back execution time measurements. These measurements can be transformed in a real performance model that is used during the deployment space exploration.

3.7 Code Generation

When a solution turns out to be feasible after the three stages, the code can be synthesised for each hardware platform in the configuration (only shown in Figure 1). This includes the generation of the C-code of the application, generation of the middleware and generation of the AUTOSAR run-time environment

---

[3] Hardware-in-the-loop is a common simulation technique used in engineering for the development and testing of embedded systems

(RTE) that is required to glue the application code and middleware code together.

4 FTG+PM: Formal Definition

In the following definitions we will provide the precise abstract syntax of the FTG+PM formalism. We will mention the relation between FTG+PM abstract and concrete syntax (as can be observed e.g., in Figure 1) whenever the abstract syntax definitions do not make that relation immediately obvious.

Definition 1. *Language and Model*
We call the set of all languages $\textsc{Form}$ *and the set of all models* $\textsc{Models}$*. A model always conforms*[4] *to a given language, formally written* $model\ conformsTo\ f$*, where* $f \in \textsc{Form}$*. The set of models that conform to a language* $f \in \textsc{Form}$ *is the set* $\textsc{Models}^f = \big\{model \in \textsc{Models} \mid model\ conformsTo\ form\big\}$*.*

Definition 2. *Transformation and Transformation Execution*
Given a set of languages $F \subseteq \textsc{Form}$*, we formally write* $t^{s_1,\ldots,s_m}_{t_1,\ldots,t_n}$ *to denote a transformation* t *where* $\{s_1,\ldots,s_m\} \in \mathcal{P}(F)$ *is the set of source languages of* t *and* $\{t_1,\ldots,t_n\} \in \mathcal{P}(F)$ *is the set of target languages of* t*. The set of all transformations for a set of languages* $F \subseteq \textsc{Form}$ *is written* $\textsc{Tr}^F$*.*

Given a set of languages $F \subseteq \textsc{Form}$*, a transformation execution of* $t^{s_1,\ldots,s_m}_{t_1,\ldots,t_n} \in \textsc{Tr}^F$ *is a computation that: receives a set of inputs models* $im_1,\ldots,im_m$ *such that* $mi_k\ conformsTo\ s_k\ (1 \leq k \leq m)$*; produces a set of outputs models* $om_1,\ldots,om_m$ *such that* $om_k\ conformsTo\ t_k\ (1 \leq k \leq n)$*. Given the above, we write* $ex\ executionOf\ t$ *to denote* ex *is an execution of* t*. The set of all executions of* t *is written* $\textsc{Exec}^t$*.*

Definition 3. *Formalism Transformation Graph (FTG)*
A formalism transformation graph is a tuple $\langle F, \tau\rangle \in \textsc{Ftg}$*, where* $F \subseteq \textsc{Form}$ *and* $\tau \subseteq \textsc{Tr}^F$*.*

In the FTG definition 3 the "graph" notion comes from the fact that languages can be seen as nodes of a graph where transformations connect the nodes via relations of input and output. In what follows we use the notation $\mathbf{V}_s$ to denote the set of variables over set s.

Definition 4. *Process Model (PM)*
Let $ftg = \langle F, \tau\rangle \in \textsc{Ftg}$*. A process model of* ftg *is a tuple* $\langle Act, Obj, CtrlNode, CtrlFlow, DataFlow, Guard, CtrlNodeType\rangle \in \textsc{Pm}^{ftg}$*, where:*

- $Act \subseteq \bigcup_{ex=\textsc{Exec}^t} \mathbf{V}_{ex}$ *such that* $t \in \tau$
- $Obj \subseteq \bigcup_{mod=\textsc{Models}^f} \mathbf{V}_{mod}$ *such that* $f \in F$
- $CtrlNode \subseteq NodeID$*, where* $NodeID$ *is a set of control node identifiers;*
- $CtrlFlow \subseteq (Act \times Act) \cup (Act \times CtrlNode) \cup (CtrlNode \times Act)$

---

[4] this is the typical conformance relation as found in the literature [23]

- $DataFlow \subseteq (Act \times Obj) \cup (Obj \times Act) \cup (Act \times Node)$
- $Guard : CtrlFlow \hookrightarrow conditionsOver(F)$[5]
- $CtrlNodeType : CtrlNode \to \{forkJoin, decision, begin, end\}$

with the following additional constraints:

- *for all* $a \in Act$ *inbound dataflow arrows carry the transformation's input models; outbound dataflow arrows carry the transformation's output models;*
- *for all* $pm \in \textsc{Pm}^{ftg}$, *CtrlNodeType is surjective regarding the restriction of the function's co-domain to* $\{begin, end\}$, *meaning that for a given process model only one* start *and only one* end *control node exist;*
- *if* $(a, n), (a', n') \in CtrlFlow$ *then* $a = a'$, *meaning only one control flow arc is allowed from each activity;*
- *if* $(a, d), (a', d') \in DataFlow$ *then* $a = a'$, *meaning only one data flow arc is allowed from each activity;*
- *if* $(d, n) \in DataFlow$ *then* $CtrlNodeType(n) = decision$;
- $Guard\big((n, n')\big)$, *where* $(n, n') \in CtrlFlow$, *is defined if and only if* $CtrlNodeType(n) = decision$.

The abstract syntax of PM in definition 4 includes the fundamental set of constructs in Activity Diagrams, as well as data flow: *Act* are action nodes (in our case placeholders for executions of transformations) and are represented as round edged rectangles; *Obj* are object nodes (in our case placeholders for instances of languages) and are represented by square edged rectangles. *CtrlNode* is a set of control nodes typed by the *CtrlNodeType* function and having the respective classical activity diagram concrete syntax. The *CtrlFlow* and *DataFlow* relations specify the edges between action, object and control nodes. Finally the *Guard* function allows defining guards for edges which are outbound of decision nodes. The constraints following the first part of definition 4 insure the well-formedness of the PM Activity Diagrams.

Definition 5. *FTG+PM*

A FTG+PM is a pair $\langle ftg, pm\rangle \in \textsc{FtgPm}$, *where* $ftg = \langle F, \tau\rangle \in \textsc{Ftg}$ *and* $pm \in \textsc{Pm}^{ftg}$ *is a process model of* ftg.

The semantics of a $\langle ftg, pm\rangle \in \textsc{FtgPm}$ is a set of traces associated with executions of the activity diagram specified in *pm*. The semantics of Activity Diagrams with data flow have been addressed by Störrle in [41] and are built by transforming UML 2.0 Activity Diagrams into Coloured Petri Nets [1], as suggested by the UML 2.0 specification [35]. The resulting traces are labelled transition systems where states hold the models available at each given moment of the development process and transitions represent transformation executions. Notice that in definition 4 action nodes and object nodes are defined as variables of transformation executions and instances of languages, respectively. However, when the traces are calculated the PM's variables are replaced by concrete transformation executions and models (see definition 2).

---

[5] we use $\hookrightarrow$ to denote a partial functions.

5 Discussion

The contribution of the paper is a framework for formalism transformation, consisting of the Formalism Transformation Graph and the Process Model. Its usefulness was illustrated by a small case study from the automotive domain. The framework allows the MDE process to be flexible. Also, insight in the domain can be gained as the FTG+PM provides modellers with a means to describe and even prescribe the MDE process. We suggest that FTG+PMs should be devised for each specific domain where MDE is used. Thus, PMs model domain-specific MDE practices. As the FTG charts all different formalisms used as well as their relationships, it can be seen as a model of all MDE artefacts.

The languages for both FTGs and PMs, as well as their relationships, were formalised in Section 4. In practice, we use a subset of UML Activity Diagrams 2.0 to express PMs. The metamodel of FTGs is a bipartite graph. Its metamodel is straightforward and was not shown in this paper because of spatial constraints. The explicit modelling of the FTG+PM and its execution semantics allow us to extend the formalisms in an MDE fashion, reaping all its benefits.

In its current state, the FTG+PM can be improved in several ways to make it more valuable. (1) The execution semantics of the PM could include an annotation mechanism to keep some information on an artifact such as author, date created, tool used, and formalism it conforms to (similar to [6]); (2) A difference can be made in the FTG between general-purpose formalisms, transformations that are likely to be reused (e.g., Petri-net to reachability graph) and domain-specific parts that are only relevant to one particular PM (e.g., C-code calibration for Autosar C-code). The general-purpose artifacts can be browsed as a library of off-the-shelf formalisms and transformations when creating a new PM; (3) Currently, all relationships in the FTG are transformations. We can classify the transformations according to their type and/or intention, e.g., model-to-model translation, verification, refinement, abstraction, code generation, simulation, etc. [25]. Generalising this further, we can add pre- and post conditions as properties to the transformations in the FTG. During the execution of the PM, these pre- and post conditions can be checked to ensure validity and correctness of the transformations. Moreover, this strategy can be combined with analytical techniques to prove properties of some of the general purpose transformations that are re-used widely.

Using the FTG+PM approach results in an ever growing centralised FTG, and an ever growing collection of PMs which can be used for empirical evaluation of current MDE techniques. We anticipate that by using data mining techniques on a collection of FTG+PMs, patterns will become apparent that can enable reuse and help designers to solve problems of ever increasing complexity. The FTG+PM framework can also be used as an enabler for tool integration where the transformations between the different model representations in the FTG can be looked up and reused within the PMs.

6 Related Work

Our work is focused on creating a platform for unifying MDE practices by defining a detailed and precise model, namely the FTG+PM, to guide the model transformation process. While FTG+PM is generic and can be applied in the development of systems in various domains, we have worked on a case study in the automotive domain to illustrate our FTG+PM and its applicability. There have been research carried out in both academia and industry on the model-driven engineering of automotive cyberphysical systems [16, 46, 37, 15]. [7] present a MDE framework based on SysWeaver for the development of AUTOSAR-compliant automotive systems.

Research related to our approach can be divided into two parts: modelling the relations between models explicitly (similar to the FTG), and describing the transformations explicitly as an MDD process (similar to the PM).

6.1 Inter-model modelling

The idea of modelling the existing relations between different processes was first introduced by Vangheluwe et al. [44] in the context of simulation. A Formalism Transformation Lattice, addressing the same goals the Formalism Transformation Graph, is introduced in [45]. The idea is further elaborated in [10], advocating AToM^3 [11] as a suitable tool for its implementation. Indeed, we use AToM^3 and its successor AToMPM excessively in the power window case study. The FTG of [10] has no formal character however and leaves transformations implicit.

Bézivin et al. introduce the concept of megamodels [6] as a global view of the considered artifacts in a system. They claim that this concept is essential in any MDD platform. Key in their approach is that not only models, but also tools and the services and operations they provide are also represented as models, with all sorts of relations in between. Megamodelling is also called *modelling in the large*. A megamodel is mainly presented as a means to store metadata (e.g., that an artifact was generated by a particular transformation or created in a particular tool, what its metamodel is, etc.). The authors state that process modelling could be achieved with megamodelling. [14] continues on megamodelling, and four different kinds of relations are presented, referring to the semantics: DecomposedIn, RepresentationOf, ElementOf, and ConformsTo.

Salay et al. introduce macromodels as a means to capture the intended purpose and a set of intended relationships (such as refinement, instantiations, refactorings, etc.) of models [38]. They model relationships between formalisms in a similar way as in megamodeling, but they allow modelling these relationships explicity as metamodels. Their goal is to improve understandability, to enforce contraints on models even before they are created, to check for consistency between models and to manage evolution of the modeling project. Similarly to megamodeling, there is no support for workflow modeling.

6.2 The MDD process

Various model transformation languages and toolsets are used in practice today, such as the OMG-standard QVT [26], Atlas Transformation Language [22], and AToM^3 [11]. Such tools are used independently for carrying out some particular purpose within MDE. However, research has shown a need for unifying MDE practices and tools [5] [31].

Process modelling has a huge following in research, resulting in many modelling languages. Recent years, most of these languages are based on π-calculus and/or Petri nets. π-calculus [29] was introduced by Robert Milner, and is based on Calculus of Communicating Systems (CCS) [28] which was developed by Milner in Parallel with Hoare's Communicating Sequential Processes (CSP) [19], all of which are prominent process calculi. Petri nets [36] were created by Carl Adam Petri as a graphical formalism to express concurrent systems. Examples of used process modelling languages that have roots in π-calculus and/or Petri nets are Business Process Model and Notation (BPMN) [34], the textual Business Process Execution Language (BPEL) [32], Coloured Petri nets [21] in e.g., CPNTools [20], Yet Another Workflow Language (YAWL) [43], Event-Driven Process Chains (EPC's) [42] and UML Activity Diagrams [35]. The XML Process Definition Language (XPDL) [47] is a well-known standard defined by the Workflow Management Coalition (WfMC) for storing visual diagrams, such as BPMN diagrams.

OMG's Software Process Engineering Metamodel (SPEM) [3], formerly known as Unified Process Model (UPM), is designed for defining the process of using different UML models in a project. SPEM is defined as a generic software process language, with generic work items having different roles. It is merely a generic framework for expressing processes, and does not include e.g., a visual concrete syntax.

Oldevik et al. [33] present a metamodel-based UML profile for model transformation and code generation. The goal of the work is provide a framework that assists transformations in the MDE lifecycle by defining activities and tasks. The paper outlines the semantics of the transformations required to map models models at a high level of abstraction (e.g. requirements) to models at the architecture and platform-specific levels.

Similar to our Process Models, Van Gorp et al. employ Activity Diagrams 1.0 to express chains of transformations [18]. Their main goals are understandability and reusability. Their notation uses regular States to denote types of models, and Object Flow States to denote transformations. The rather preliminary language uses Synchronisation Bars as well. They are used to denote synchronous execution (in case of multiple outputs of Synchronisation Bars), as well as multiple transformation inputs/outputs for a transformation (in case of multiple inputs of Synchronisation Bars). The language does not include decision diamonds and has no precise semantics, but is rather used as a documentation means.

We ultimately chose Activity Diagrams 2.0 as our formalism for modelling processes for three reasons: the formalism is well-known, especially in the field of modelling, the formalism is well-supported by general tools, and it allows us to model both control flow and data flow.

7 Conclusion

In this paper, we presented a framework for carrying out formalism transformations within MDE. We proposed the Formalism Transformation Graph (FTG) and the Process Model (PM) to guide the model-driven development process. We introduced the FTG+PM framework, composed of the FTG and its complement, the PM. The FTG comprises formalisms as nodes and transformations as edges, and shows the different languages that need to be used at each stage of development. Meta-modelling and model transformation are the basis of the FTG. The FTG explicitly models the relations between requirements, design, simulation, verification, and deployment languages. The transformations are depicted as Activity Diagram 2.0 actions, and the control flow and the data flow between each transformation action are detailed in the Process Model (PM).

We have applied FTG+PM on a non-trivial case study of the design of an automotive power window controller. We have constructed the FTG and PM for the target domain in the FTG+PM language. Following requirements elicitation and specification using the SysML use case diagram and requirements diagram formalisms, we have defined domain specific languages that allow modelling of the main components of the control system: the environment, the plant, and the control. The DSLs are transformed to Petri nets to carry out reachability analysis on the one hand, and to a hybrid simulation formalism (composed of a continuous time formalism Causal Block Diagrams, and a discrete-event formalism, Statecharts) to ensure that system constraints are satisfied. After successful safety analysis and simulation, the control model in the Statecharts formalism is mapped on to deployment models. We have used the AUTOSAR middleware to deploy our software onto a hardware architecture. The deployment is a multiphase process beginning with the generation of a calibration infrastructure which feeds into a performance model, followed by an intial architecture deployment (in C-code), bin packing analysis and schedulability analysis to check that performance constraints are being met, and finally simulation using the DEVS formalism. Additionally, timing requirements (represented using the TIMMO language) are derived from the inital requirements diagrams, and integrated and checked during deployment. Once the simulation outputs an acceptable trace, the deployment models are transformed to middleware code and RTE code.

The FTG and PM we have introduced can be adapted for use in various domains. It provides a complete model-driven process that is based on meta-modelling, multi-abstraction and multi-formalism modelling, and model transformation. We plan on extending this work and adapting the FTG+PM for feature-oriented software development of (software) product lines.

References

1. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use (Volume 1)*, volume 1 of *EATCS Series*. Springer Verlag, 2003.
2. ATLAS transformation language, 2008. `http://www.eclipse.org/m2m/atl/`.
3. SPEM version 2.0, OMG document number formal/2008-04-0, april 2008.
4. AUTOSAR. Official webpage. `http://www.autosar.org`, 2010.
5. Jean Bézivin and Erwan Breton. Applying the basic principles of model engineering to the field of process engineering. *UPGRADE: European Journal for the Informatics Professional*, 27-33, 2004.
6. Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE*, 2004.
7. G. Bhatia, K. Lakshmanan, and R. Rajkumar. An End-to-End Integration Framework for Automotive Cyber-Physical Systems Using SysWeaver. In *AVICPS 2010*, pages 23–30, 2010.
8. H. Blom, R. Johansson, and H. Lönn. Annotation with Timing Constraints in the Context of EAST-ADL2 and AUTOSAR, the Timing Augmented Description Language. In *STANDRTS'09*, 2009.
9. Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wils. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.
10. J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Soft. and Sys. Mod.*, 3(3):194–209, 2004.
11. Juan de Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In *FASE '02*, pages 174–188. Springer-Verlag, 2002.
12. Richard C. Dorf. *Modern Control Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 12th edition, 2011.
13. Grégoire Dupe, Mariano Belaunde, Romain Perruchon, Héléna Besnard, Florian Guillard, and Vivian Oliveres. SmartQVT. `http://smartqvt.elibel.tm.fr/`.
14. Jean-Marie Favre. Megamodelling and etymology. In *Transformation Techniques in Software Engineering*, Dagstuhl Seminar Proceedings, 2006.
15. J. Friedman and J. Ghidella. Using model-based design for automotive systems engineering - requirements analysis of the power window example. *SAE*, 2006.
16. Z. Gao, H. Xia, and G. Dai. A model-based software development method for automotive cyber-physical systems. *Comput. Sci. Inf. Syst*, 8(4):1277–1301, 2011.
17. H. Giese, T. Levendovszky, and H. Vangheluwe. Summary of workshop on multi-paradigm modeling: concepts and tools. In *MoDELS'06*, pages 252–262, 2006.
18. P. Van Gorp, D. Janssens, and T. Gardner. Write once, deploy n: A performance oriented mda case study. In *EDOC*, pages 123–134. IEEE Computer Society, 2004.
19. C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
20. K. Jensen, L. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, 2007.
21. Kurt Jensen. *Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use: volume 1*. Springer-Verlag, London, UK, 1996.
22. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.
23. Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

24. L. Lucio, J. Denil, and H. Vangheluwe. An overview of model transformations for a simple automotive power window. Technical report, McGill University, 2012.
25. T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.
26. Meta object facility (MOF) 2.0 query/view/transformation (QVT), OMG document number formal/2011-01-01, available from www.omg.org, 2011. Version 1.1.
27. Metacase. Domain-Specific Modeling with MetaEdit+: 10 times faster than UML, 2009.
28. R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
29. R. Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, New York, NY, USA, 1999.
30. W. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the GEF and the EMF*. IBM RedBooks, February 2004.
31. Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation*, 80(9):433–450, 2004.
32. OASIS. Web Services Business Process Execution Language Version 2.0, Apr. 2007.
33. J. Oldevik, A. Solberg, B. Elvesæter, and A. Berre. Framework for Model Transformation and Code Generation. In *EDOC*, pages 181–189, 2002.
34. OMG. Business Process Model and Notation Version 2.0, Jan. 2011.
35. OMG. Unified Modeling Language (UML) 2.4.1 Superstructure, Aug. 2011.
36. J. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.
37. S. Prabhu and P. Mosterman. Model-based design of a power window system: Modeling, simulation and validation. In *IMAC-XXII*, 2004.
38. R. Salay, J. Mylopoulos, and S. Easterbrook. Managing models through macromodeling. In *ASE*, pages 447–450. IEEE, 2008.
39. A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6):23–33, 2001.
40. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20:42–45, September 2003.
41. H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. In *ENTCS*, pages 35–52. Elsevier, 2004.
42. W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639 – 650, 1999.
43. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245 – 275, 2005.
44. Hans Vangheluwe, Bo Hu Li, Y. V. Reddy, and G. C. Vansteenkiste. A framework for concurrent simulation engineering. In *Simulation in Concurrent Engineering*, pages 50–55. SCS, 1993.
45. Hans Vangheluwe and G. C. Vansteenkiste. A multi-paradigm modeling and simulation methodology : formalisms and languages. In *ESS'96*, pages 168–172, 1996.
46. S. Wang. Model transformation for high-integrity software development in derivative vehicle control system design. In *HASE*, pages 227–234, 2007.
47. WfMC. XML process definition language (XPDL), October 2005.