FTG+PM: An Integrated Framework for Investigating Model Transformation Chains

Levi Lúcio[1], Sadaf Mustafiz[1], Joachim Denil[2,1], Hans Vangheluwe[2,1], and Maris Jukss[1]

[1] School of Computer Science, McGill University, Canada
{levi,sadaf,joachim.denil,hv,mjukss}@cs.mcgill.ca
[2] University of Antwerp, Belgium
{joachim.denil,hans.vangheluwe}@ua.ac.be

Abstract. In this paper, we describe our ongoing work on *model transformation chains*. Model transformation chains refer to the sequences of model transformations in Model Driven Engineering (MDE). The transformations represent and formalise typical model/software engineering activities, and their chaining is the natural composition of such activities. Model transformation chains found in industrial practice vary widely, depending on the specific domain they are used in. By explicitly modelling development activities, these activities can be analysed and the MDE process may be improved. As a step towards such analyses, we propose an integrated framework to describe all the artifacts involved in model transformation chains, as well as the means to execute "enact" those chains. We describe the Formalism Transformation Graph + Process Model (FTG+PM) which is at the heart of our framework in detail.

1 Introduction

Model Driven Engineering (MDE) is currently the mainstream top-down approach to software development. The philosophy behind MDE is that software development should start by building *domain specific* structural and behavioral models of the system under development. By *domain specific* we mean that initially models of the system should be described in a language close to the domain being tackled. During the software development process those models are then improved, augmented and refined by the application of model transformations – possibly with the automatic or manual injection of additional information.

Model transformations have been called the *heart and soul* of MDE [1]. Chaining model transformations is a natural step in MDE as such chains allow describing the composition of activities in software construction and provide explicit means for MDE automation. However, to the best of our knowledge little work is devoted to understanding the underlying structure of such chains when they are used in domain specific software development. This work is crucial for the following (non-exhaustive) list of reasons:

– **Reuse**: Model transformation chains are typically devoted to building software within certain domains. In this paper, we provide an example of the usage of model transformation chains for building automotive software. As in traditional software development, the modularity and possibility of reuse of such chains is extremely relevant from an engineering viewpoint. It seems natural that subsets of a transformation chain developed for a given software engineering purpose can be reused without much changes for a similar engineering purpose. Moreover, by identifying and classifying subsets of transformation chains responsible for high level activities in domain specific software development (e.g. requirements development, domain-specific design, verification, simulation, analysis, calibration, deployment, code generation, execution, etc), it is possible to achieve a finer level of understanding and control of such activities – in a domain specific or in a more general context;

- **Traceability**: Traceability is increasingly required in software development at the stakeholder level (e.g. to ensure a given requirement has been implemented in the system), but also at the software development level (e.g. to ensure traceability as high level models are refined along the development process). Because transformation chains explicitly model the relations between the several steps of an MDE process, traceability is a natural consequence of using such chains;
- **Certification**: Finally, and possibly most importantly, by having an explicit representation of such transformation chains and the models (and metamodels) they work on, the certification of such processes becomes possible. In certain domains such as embedded systems, automotive or aerospace, strict norms exist to ensure each step in software production is performed correctly and is properly documented. A large effort has been devoted in the last two decades to developing verification methods for software. The MDE community is now missing studies on how and when those techniques should be applied, but also how they can be composed in a meaningful way. Again, model transformation chains are the ideal context to study the usage and utility of such verification methods for software certification in MDE.

Several studies such as [2,3,4,5,6,7,8], among others, have addressed model transformation chains. However, to perform an investigation on the nature and pragmatic uses of transformation chains we require an environment where all the artifacts involved in such chains are explicitly formalized, easily accessible and easily manipulated. The majority of the approaches in the literature dealing with transformation chains are concerned with automated execution. The explicit and integrated representation of all artifacts involved in model transformation chains in a way that makes them amenable to the formal study of those chains' characteristics is typically less of a concern. In order to address this issue and to have a solid basis to study the issues mentioned above, we need a framework allowing the modelling of model transformation chains that addresses the following requirements:

1. An explicit representation of both the languages used in the model transformation chains and the relations between those languages should be provided;
2. An explicit representation of the individual model transformations should be available and the means to execute those transformations should exist;

3. Explicit process modelling of MDE activities should be possible such that transformation chains can be built;
4. Automatic execution of transformation chains should be possible. In order to study the execution of transformation chains and which parts of those chains should be performed manually, we require that a model transformation chain execution engine exists.

In order to address these requirements, we propose in this paper the FTG+PM framework. The proposed framework is completely supported by our tool AToMPM, A Tool for Multi-Paradigm Modelling [9], which allows explicit modelling of and access to, all used artifacts.

This paper is organised as follows: Sect. 2 provides background information on meta-modelling, model transformation, and our tooling environment. In Sect. 3 we introduce our running example, the *power window case study.* Section 4 introduces the FTG+PM framework. Section 5 presents the the explicit execution semantics of the FTG+PM. Section 6 describes in detail an automotive power window case study and by doing so illustrates the artifacts involved in a model transformation chain. Section 7 discusses related work. Finally, Sect. 8 draws some conclusions on how the FTG+PM addresses the aforementioned requirements and proposes future studies on model transformation chains.

2 Background

Within the context of this paper we have chosen to follow the terminology as presented in [10]. A *model* is completely described by its abstract syntax (its structure), concrete syntax (its visualisation) and semantics (its unique and precise meaning). A *language* (also called *formalism*) is a possibly infinite set of (abstract syntax) models. This set can be concisely described by means of a grammar or a metamodel. No semantics or concrete syntax is given to these models. Several such languages, called *metamodels*, are used to describe families of models of computational artifacts that share the same abstraction concerns. Each *metamodel* is a language that may have many *model* instantiations.

Domain Specific Modelling (DSM) captures the fact that certain languages or classes of languages, called Domain Specific Languages (DSLs) are appropriate for expressing models in certain domains.

Model transformations involve the mapping of source models in one or more formalisms to target models in one or more formalisms using a set of transformation rules.

In this work, we use rule-based graph transformation as the means for model transformation [11]. This requires (meta-)models to be stored as graphs, thus allowing model manipulations to be defined as graph grammars.

In our work, we have used *AToMPM* [9]*, A Tool for Multi-Paradigm Modelling,* to build metamodels, transformations, and execution support for the FTG. AToMPM (the successor of AToM^3[12]) rigorously applies the "model and conforming meta-model" workflow to all facets of domain specific modelling. It allows modelling of language syntax (abstract and concrete) and semantics. The tool supports rule-based graph transformations and pre- and post-condition pattern languages to allow specification of model transformations. AToMPM runs on a web browser and provides support for real-time, distributed collaboration.

3 The Power Window Case Study

In order to explain how our FTG+PM framework addresses the requirements stated in Sect. 1 we will use a running example. In Fig. 1 we show a slice of the FTG+PM we have built for developing the power window control software. The power window FTG+PM was built based on our experiences with developing automotive software. Further details on this case study can be found in [13,14].

A power window is basically an electrically powered window. 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. When given the task to build the control system for a power window, a software engineer considers several variables, such as:

(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 along the same lines as that presented by Mosterman and Vangheluwe in [15]. According to control theory [16], 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*.

The FTG+PM slice in Fig. 1 presents the design and verification part of developing the power window software. The case study begins with three domain-specific languages built for the modelling of power windows (*PlantDSL*, *EnvDSL* and *ControlDSL* in the FTG part of Fig. 1, allowing respectively modeling the *plant*, *environment* and *controller* for a power window), plus a network language (not shown in Fig. 1) that allows the connection of the components defined in those DSLs. Those domain specific components are separately transformed into modular Petri nets (*EncapsulatedPetriNet* in the FTG part of Fig. 1). When all the modular Petri nets have been built, they are composed into a single Petri net (*PetriNet* in the FTG part of Fig. 1). This Petri net can then be used to verify that the system cannot enter a non-safe state. While the left side of Fig. 1 presents the FTG part of the model detailing the required formalisms and transformations, the right side of Fig. 1 shows how executions of those transformations are chained. Note also that in Fig. 1 dotted elements *PlantToPN*, *EnvToPN* and *ControlToPN* denote automatic transformations, while other elements without dots denote manual ones. The following section elaborates on the syntax of the FTG+PM language.

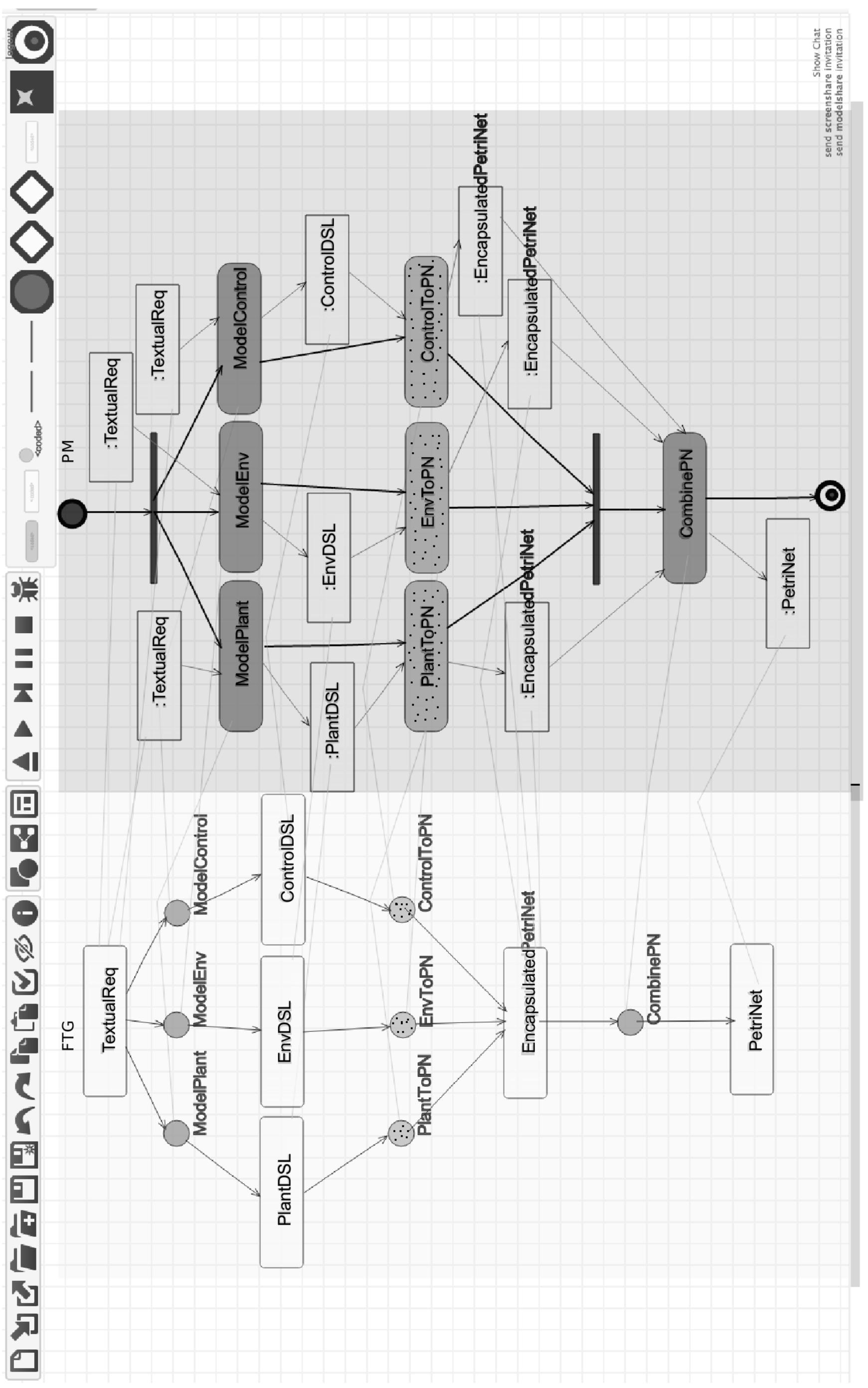

Fig. 1. Example FTG+PM Model

4 The FTG+PM Language

The FTG+PM language is defined using two sub-languages: the Formalism Transformation Graph (FTG) language and a Process Model (PM) language. We give a brief overview of FTG+PM in this section. The formalization of the language along with further details on the framework can be found in [17]. A unified metamodel of the FTG+PM language is shown in Fig 2.

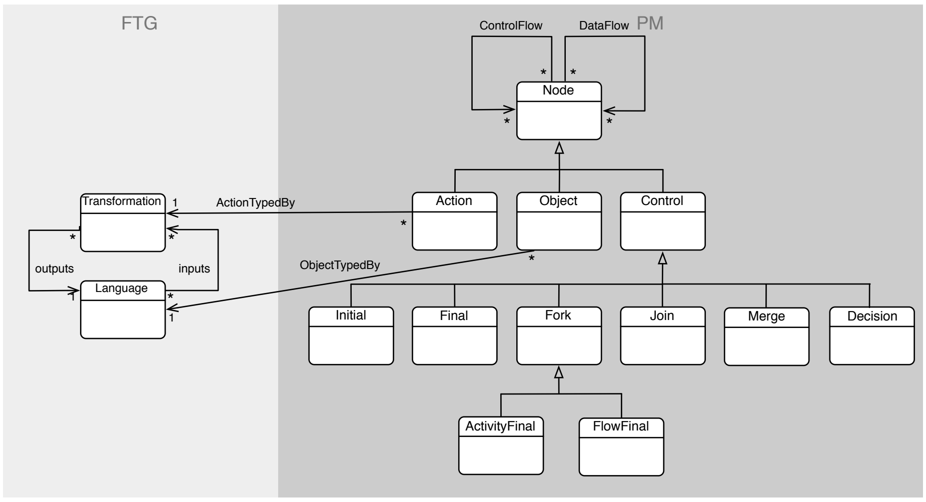

Fig. 2. Formalism Transformation Graph and Process Model (FTG+PM) Metamodel

The *Formalism Transformation Graph (FTG)* is a hypergraph with *languages* as nodes and *transformations* as edges. It lays down the relationships among the multitude of languages and transformations used for the development of a particular system or systems within a domain. The framework takes into account the heterogeneous nature of the MDE process, and integrates the MDE paradigms: multi-abstraction, multi-formalism, and metamodelling. The languages at each level in the FTG are used to represent and model knowledge at different levels of abstraction starting from requirements to code synthesis. Depending on the activity involved, we build our FTG by choosing the most appropriate formalism based on the nature of the problem and the intention: discrete-event formalisms, continuous time formalisms, hybrid formalisms, or others. All the languages in the FTG are metamodelled, and the transformations are specified using rule-based graph grammars. Languages in the FTG are denoted by labelled rectangles, and transformations are denoted by labelled circles on edges. The incoming edges show the source languages of the transformation, and the outgoing edges point to the target languages. Fig 1 (discussed in detail in Sect. 6) shows a slice of a FTG+PM model for the automotive domain (complete model presented in [14]), and describes a part of the artifacts and the process necessary to build software to control power windows of automobiles. The FTG model may include self loops to languages (for example, when a transformation is endogenous in nature).

The *Process Model (PM)* (see sample PM in Fig 1 highlighted in gray) is used in conjunction with the FTG to model the MDE process. Having a process model integrated with the FTG allows us to precisely and in detail model the MDE process we follow, and to provide execution support for it when needed. The PM exhaustively describes the control flow and data flow in the MDE process. Our process model is a subset of the UML 2.0 activity diagram metamodel. In the PM language, the labelled roundtangles (actions) in the Activity Diagram correspond to executions of the transformations declared within the FTG. This typing relation is made explicit in the FTG+PM model by the thin horizontal links connecting the action nodes in the PM to the transformation elements in the FTG. Labelled rectangles (object nodes) in the PM correspond to models that are consumed or produced by actions. A model is an instance of a language declared in the FTG part of the model with the same label. This typing relation is again made explicit by horizontal links connecting the object nodes to the language elements in the FTG. Notice that in a PM model thin edges denote data flow, while thick edges denote control flow. Notice also that for each model input and output edge of a PM action a corresponding edge exists for the transformation typing it on the FTG side. The input and output models of an action are typed according to the input and output languages of the FTG transformation that types that action. Finally, the join and fork Activity Diagram flow constructs represented as horizontal bars, allow us to represent concurrent activities.

The FTG defines a set of transformations and the PM describes the chaining of the transformations and the execution order for a particular intent. The FTG+PM can thus be considered to be a model transformation chaining language for describing the composition of transformations by defining their order of execution, source and target model types, and the relationships and dependencies among them.

Various business process modelling or workflow languages exist in the literature. Our intention is to model the MDE process as a chain of model transformations rather than a business process with models as first class artifacts and with model transformations as the core of the approach, hence we have chosen to use UML 2.0 activity diagrams for our purpose. In addition, UML 2.0 is a standard in the MDE community, and our tool, AToMPM (A Tool for Multi Paradigm Modelling) [9] also provides support for UML. Our framework is supported by AToMPM for creating metamodels, describing graph transformations, and for building execution support for the FTG.

5 FTG+PM Semantics: Transformation and Tool Support

The proposed FTG+PM language is implemented in our AToMPM tool. AToMPM contains its own transformation language. Transformations and transformation rules, in AToMPM, are treated as normal models conforming to an appropriate meta-model. Transformation rules, consisting of a left hand side (LHS), a right hand side (RHS) and a set of negative application conditions (NAC), are tried in an order given by a rule scheduling model, in this case described in a finite state automaton-like formalism. Since transformation rules and their scheduling are explicitly modelled within AToMPM using appropriate meta-models, defining higher-order transformations is straightforward.

To execute a FTG+PM model, we transform the PM to the native transformation scheduling language of AToMPM. The result of the transformation of the power window FTG+PM shown in Fig. 1 to the native AToMPM transformation language is depicted in Fig. 3. A PM action which is mapped to a transformation can be either automatic (e.g. see dotted elements *PlantToPN* etc. in Fig. 1) or manual (other elements in Fig. 1 without dots). Manual transformations are not implemented using graph transformations, but involve actions in which the output models need to be created by the user(s).

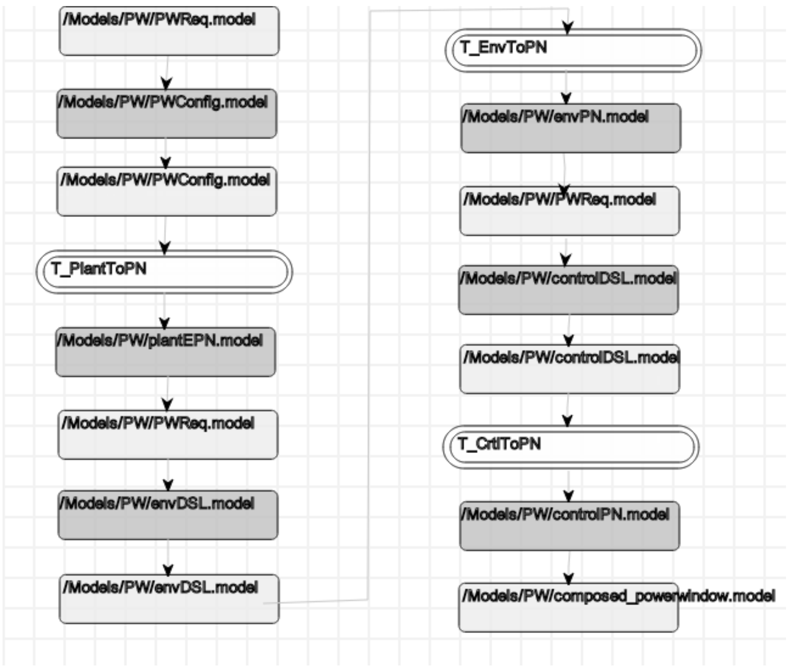
/Models/PW/PWReq.model
/Models/PW/PWConfig.model
/Models/PW/PWConfig.model
T_PlantToPN
/Models/PW/plantEPN.model
/Models/PW/PWReq.model
/Models/PW/envDSL.model
/Models/PW/envDSL.model
T_EnvToPN
/Models/PW/envPN.model
/Models/PW/PWReq.model
/Models/PW/controlDSL.model
/Models/PW/controlDSL.model
T_CrtlToPN
/Models/PW/controlPN.model
/Models/PW/composed_powerwindow.model

Fig. 3. The resulting transformation model of the example FTG+PM

The transformation schedule is created as follows:

(1) a PM *Action* node tagged as *automatic* corresponds to the execution of a transformation defined in the FTG *Transformation* node typing it;
(2) transformations are scheduled according to the control-flow defined in the PM.

An example rule of this transformation from FTG+PM into AToMPM's transformation scheduling language is shown in Fig. 4. Note that the LHS of a rule matches a pattern in the input model including a PM *Action* (round-tangle) typed by a FTG *Transformation* (circle), while the RHS rewrites it by building the scheduling of the transformation execution as a double round-tangle (a composite transformation application in AToMPM's rule scheduling language). The double round-tangle is then used to execute this transformation within the AToMPM environment. For example, the *PlantToPN* action in Fig. 1 is mapped to the transformation *T_PlantToPN* (inside the double round-tangle node) in Fig. 3.

The scheduling language additionally includes rectangular nodes corresponding to the execution of a single transformation step to handle opening of input models (shaded as */Models/PW/PWReq.model* in Fig. 3) or writing (includes editing and saving) of output models (shaded as */Models/PW/PWConfig.model* in Fig. 3), and control flow arrows to impose the ordering of the scheduling of the transformations.

When executing a FTG+PM model, the input of a scheduled transformation depends on whether there are incoming dataflow arrows:

(a) if there are incoming dataflow arrows into the action node, for each of these dataflow arrows a transformation step is created that opens the specified input model in the appropriate formalism in the current canvas. The transformation rules that open the specified models are scheduled before the execution of the transformation defined by the action node;
(b) If there is no incoming data flow arrow, the result of the previous transformation (present on AToMPM's modelling canvas) is used as the input.

A similar solution is used for the output of an action:
(a) when a dataflow arrow emanates from an action node, a transformation step is created to save the target model (specified in the location by the object node) and clears the modelling canvas. The transformation step is scheduled after the transformation defined by the action node;
(b) If no dataflow arrow exits the node, the canvas is not cleared.

Manual transformations (such as *ModelPlant* highlighted in dark gray in Fig 1) are not mapped to a transformation, i.e. a double round-tangle node, in the resulting schedule. They are mapped to transformation steps that first open the input models and then to transformation steps that write/save the output models. One transformation step corresponds to one open/save model and one or more associated formalisms. For instance, the *ModelPlant* action is mapped to a pair of transformation steps in Fig. 3:

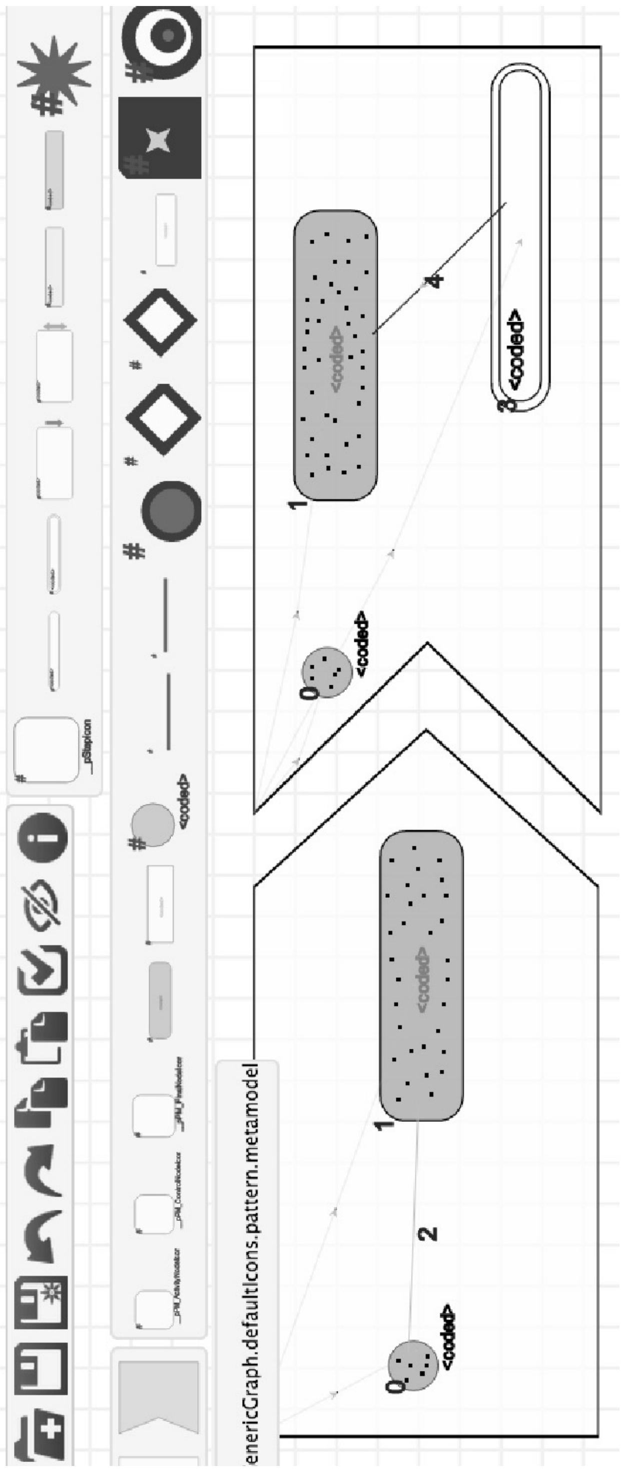

Fig. 4. Transformation Rule to Map an Action Node to a Transformation

(1) a step that opens the input requirements model, `/Models/PW/PWReq.model`, which is a model instance of *TextualReq*;
(2) a step that writes `/Models/PW/PWConfig.model`, the output plant configuration model, which is an instance of *PlantDSL*.

In case of multiple input and output models, a transformation step is created in the schedule corresponding to each open and write step. When output model(s) are produced by manual transformations, a new AToMPM window is spawned for each output model, which loads the model if it already exists (to allow for further editing) or opens an empty canvas with the formalism toolbars loaded otherwise. Once the user is done creating or modifying the model, a button needs to be pressed to save the model and to return to the parent AToMPM window where automatic transformation resumes.

In the current implementation, there is no support for the (semi-)parallel execution of fork and join nodes since the current transformation language in AToMPM does not allow this. Instead, the transformation towards the AToMPM transformation language makes sequential the different branches between the joins and the forks. This is done in the same way as described in [18] where a marker is made at the top of the fork. Another marker is used to follow the chain until the join node is found. Afterwards the full branch is scheduled before the join node. This is done until all branches are made sequential.

When nesting occurs, the inner fork/join pairs are made sequential first.

Since the canvas can be used as the input for the next action node, the state of the canvas has to be saved before the fork node. This is done by inserting an object node, connected to the action node before the fork node. The output goes to the first actions of each of the branches after the fork. At the last action of each branch, a similar object node is inserted that is connected to the first action after join node.

Because all models are saved and closed after the action node has executed and reloaded before starting a new action, the sequential process model preserves the original semantics.

6 Languages and Transformations in the Power Window Case Study

In this section, we present some of the languages and transformations that allow building and executing the transformation chains in the power window FTG+PM. Note that all the metamodels, models and transformations we present in Sections 6.1 and 6.2 have been built and are readable and/or executable using the AToMPM modelling environment. Note also that in the sections that follow the metamodels, models and transformations are not explained in complete detail, as the goal of their presentation in this paper is to illustrate the usage of the FTG+PM, rather than the case study itself. Again, for further details on the models presented in the sections that follow we refer the reader to [13].

6.1 Building the Domain Specific Languages

The design and verification part of the power window FTG+PM in Fig. 1 makes use of several domain specific languages (DSLs) for defining *controller*, the *plant* and the *environment* models.

Due to space reasons, we only present in this text the *plant* DSL which allows the specification of the hardware necessary for a given power window configuration.

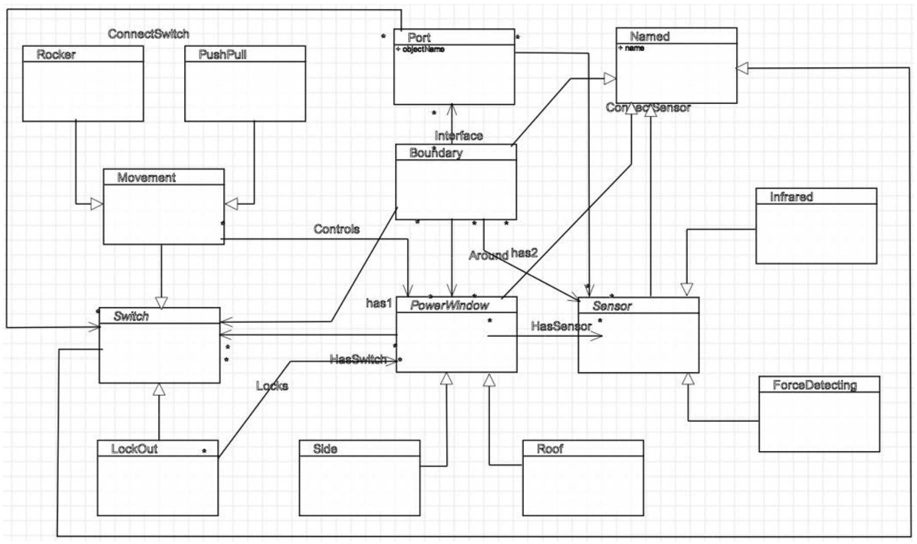

Fig. 5. Plant DSL Metamodel

In Fig. 5, the metamodel of the *plant* DSL can be observed. The main class of the language is the *PowerWindow* class, which is abstract and can be instantiated as a *Side* window or a *Roof* window. A physical power window includes a set of switches of two kinds: *Lockout* switches allow removing control from other power windows in the car (as specified by the *controls* association); *Rocker* or *PushPull* switches allow controlling window movement. Finally, a power window may also have sensors of types *Infrared* or *ForceDetecting* for detecting if an object is blocking the window from going up.

In Fig. 6, we present a model instance of the Plant DSL, where a configuration of two power windows of an automobile is described. The model includes a *driver* and a *passenger* power window, where the driver's window has three buttons: a pushpull button for controlling the driver's window, a pushpull button for controlling the passenger's window, and a lockout switch for disabling/enabling the control of the passenger's window. The passenger's window includes a rocker button and a infrared sensor meaning the window automatically stops rolling up when an object obstructs its path.

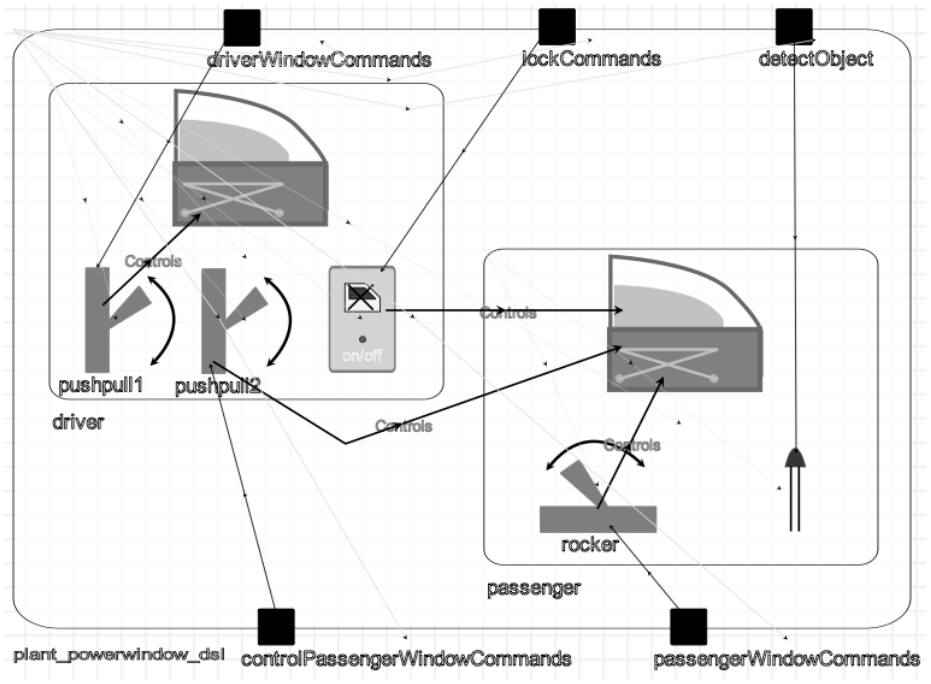

Fig. 6. Plant DSL: Example Model

6.2 Transformations

From Domain Specific Models to Modular Petri Nets. Two types of modular Petri nets are generated from the Plant DSL model by the *PlantToPN* transformation in Fig. 1, depending on the power window configuration. In Fig. 7, the Petri net modelling the discrete behavior of a power window with an obstacle detecting sensor can be observed. During operation the window can either be at the *bottom* of the frame (*bot* place, meaning the window is completely open), somewhere in the *middle* of the frame (*mid* place, meaning the window is partially open), or at the top of the frame (*top* place, meaning the window is closed). Additional places in Fig. 7 (*midDetObj*, *topDetFrame* and *danger*) are used to model object detection during window operation. The modular Petri net in Fig. 7 also includes ports (having as concrete syntax black squares) for synchronisation with other modular Petri nets. An example rule of the *PlantToPN* transformation in Fig. 1 is shown in Fig. 8. This particular rule builds the behavior of a power window without obstacle detection. Notice that the negative application condition of the rule (inside the dashed square) prevents the power window that is matched by the LHS of the rule from having a sensor.

Due to space constraints, we are unable to present here the similar transformations into modular Petri nets defined for both the control and environment models (called *EnvToPN* and *ControlToPN* in Fig. 1).

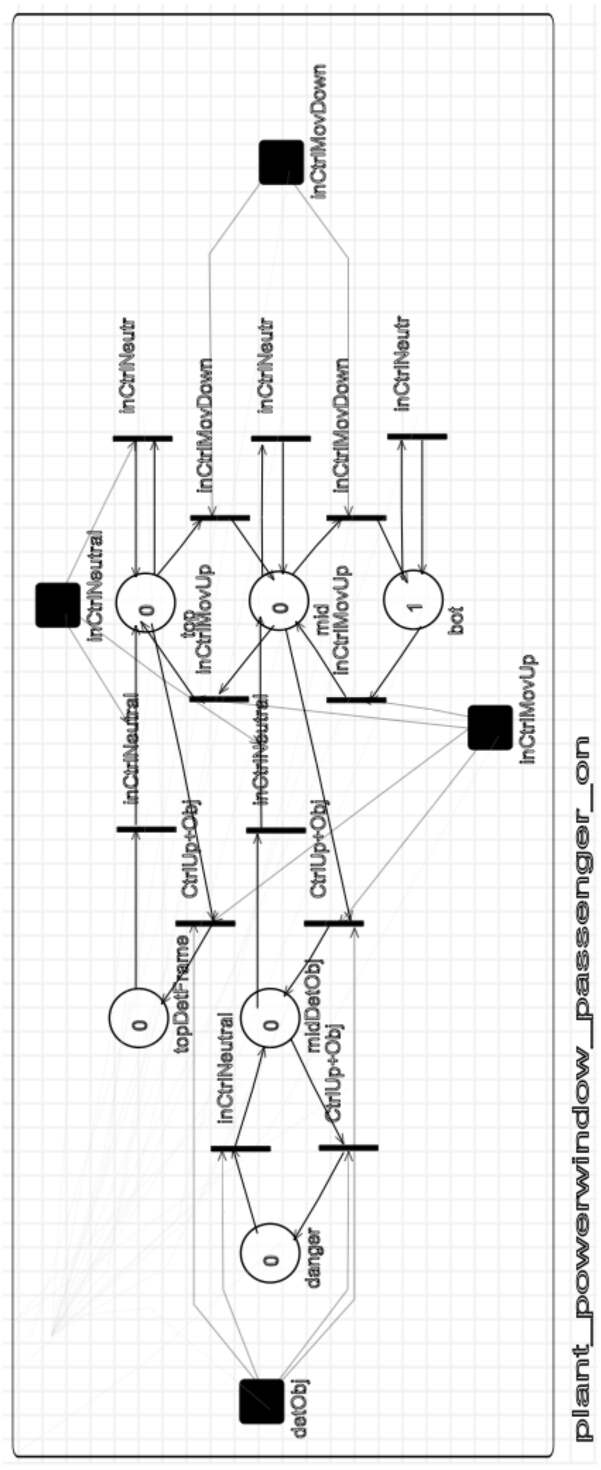

Fig. 7. Transformed Passenger Window Plant Model

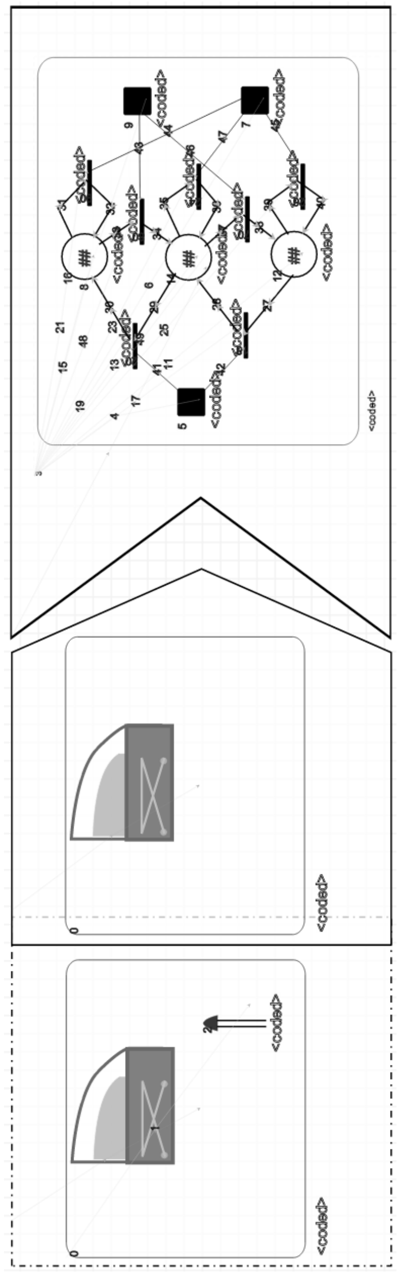

Fig. 8. Transformation Rule for Building a Petri Net Representation of a Power Window without Obstacle Sensor Plant

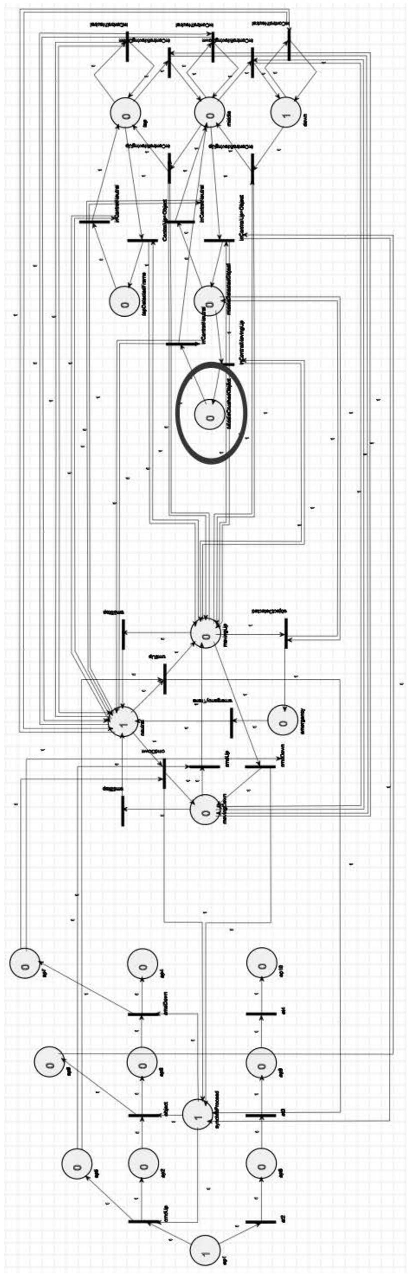

Fig. 9. Composed Petri Net Model for the Power Window Control Software

Composition of the Modular Petri Nets. Once the environment, plant, and control models are transformed to the modular Petri nets, it is necessary to compose those models. This last transformation, called *CombinePN* in Fig. 1, allows to manually[1] build the complete Petri net of the power window example using the produced modular Petri nets and an additional network model (not shown here). This composed Petri net is an instance of the PetriNet formalism in the FTG part of Fig. 1. An example of such a model (produced from our example models in Fig. 6 and Fig. 7) can be (partially) observed in Fig. 9. This composed Petri net is used for the validation of the safety requirements of the power window. In particular we have used it to automatically check that a state where an object obstructing the window has been detected and the window is still going up is never reached. In this state a token exists in the "danger" place in the EncapsulatedPetriNet model in Fig. 7. This place can be found in the rightmost subnet of the composed model in Fig. 9, highlighted by a red ellipse. Note additionally that in the transformation chain found in the complete power window FTG+PM defined in [14], the Petri net verification step is itself built as a transformation.

7 Related Work

We consider different approaches for the composition of model transformation chains. We have looked at work which have applied mega-modelling concepts and/or process modelling concepts in their approach. A *megamodel* is a conceptual framework used to reason about MDE and represents the global view of the considered artifacts (models, metamodels, and other global entities) in a system and the relationships between them [19,20]. 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.

The approaches are compared on a number of properties. The first criteria is whether the approach uses mega-modelling and therefore has an explicit representation of the modelling languages and relations between the languages by means of transformation definitions. The second is whether the approach allows the composition of chains by means of an explicit representation of the process. Finally, we consider both automatic transformations where the execution of the transformation is completely automated and manual transformations where a modelling environment is setup in the defined language(s). Table 1 shows the comparison of the different approaches.

Most approaches allow for the data-flow composition of model transformations where input and output relations of the transformations are used to chain different transformations. The control-flow of these approaches is inferred from this data-flow composition. Oldevik proposes a framework for the data-flow composition of transformations in [2]. It uses UML activities like our FTG+PM to model these relations, though control flow is not taken into account. A definition

[1] We are currently building the transformation to automatically execute this composition.

Table 1. Comparison of the approaches (supports (✓), does not support (x) , unknown/unclear (∼))

Tool	Explicit Megamodel	Explicit Process Model		Transformations	
		Control Flow	Data Flow	Automatic	Manual
Oldevik et al. [2]	✓	x	✓	✓	∼
Vanhooff et al. [3]	✓	x	✓	✓	x
UniTI [4]	✓	x	✓	✓	x
TraCo [5]	✓	x	✓	✓	x
Wagelaar [6]	x	x	✓	✓	x
MoTCoF [7]	∼	x	✓	✓	x
Wires* [21]	x	x	✓	✓	x
transML [22]	✓	∼	✓	✓	x
Epsilon [23]	∼	x	✓	✓	∼
MCC [8]	x	x	x	✓	x
Aldazabal et al. [24]	x	✓	✓	✓	∼
Diaw et al. [25]	✓	x	✓	✓	∼
FTG+PM	✓	✓	✓	✓	✓

for manual transformations is present, though it is not described how the framework copes with these transformation types. In [3], a data-flow composition of transformation framework is presented similar to the UniTI framework [4]. The concepts of these frameworks are extended by the TraCo framework [5] where additional validation checks are performed on the composition of the transformations. Wagelaar [6] presents a DSL for the composition of transformations. The models are transformed to ANT scripts for execution. Seibel et al. present the MoTCoF framework [7] for the data-flow and context composition of model transformations. The meta-model of the approach is not shown, but most likely an explicit megamodel is present. Wires* [21] provides a graphical language for the orchestration of ATL model transformations. It has modelling elements for complex data-flow for example decision nodes, parallel execution and support for loops. It does not however take manual activities into account. The transML framework [22] is created for transformations in the 'large'. It provides meta-models for requirements, analysis, architecture and testing of transformations. The tool supports data-flow chaining of transformations by transforming to ANT-tasks. The Epsilon Framework, presented in [23], provides a model management framework where ANT-tasks can be used to build chains of transformations. It is not clear if the Generic Model Manipulation Task can be used for the loading of a modelling environment though models can be loaded and stored using ANT tasks. Finally, Kleppe proposes a scripting language MDA Control Center (MCC) [8] for combining multiple transformations in sequence and in parallel.

In the process modelling community, frameworks for MDE are proposed as well, though these usually do not focus on transformation chaining, for example [26,27]. Two examples however do take transformation chaining into account.

In [24], Aldazabal et al. present a framework for tool integration where transformations can be chained. The process is modelled in SPEM or BPMN (Business Process Modelling Notation) and is transformed to BPEL (Business Process Execution Language) for execution support. They do not however have a megamodel to validate input-output relations. In [25], Diaw et al. present an adaptation of SPEM [28] for the use in an MDE context. The composition is a data-flow composition like most transformation chaining approaches discussed above. Both frameworks allow the modelling of manual activities, though it is not clear how the frameworks handle these manual activities.

Our approach, combines the explicit modelling of the languages and transformations (megamodel) together with a process model that supports complex control-flow constructs. This allows the modelling of non-linear transformation chains for building complex applications. Transformations can either be executed automatically or require manual intervention. In the manual case the framework opens a modelling environment for the activity and continues the process when the activity is finished. The explicit modelling of all the components allows to reason about these complex chains of transformations.

8 Conclusion and Future Work

In this paper, we have presented a framework for explicitly describing model transformation chains within MDE. We have introduced the FTG+PM language, composed of the Formalism Transformation Graph (FTG) and its complement, the Process Model (PM). The building blocks of the FTG are formalisms (nodes in the graph) and transformations (edges in the graph). The FTG describes the different languages that can be used at each stage of model development. The transformations model development activities, and the control flow and data flow between each transformation action are explicitly modelled in the PM.

In its current form, the FTG+PM framework satisfies the requirements stated in Sect. 1. We have explicitly described the abstract and concrete syntax of the FTG+PM language by metamodelling them in our tool, AToMPM. In addition, the syntax of each of the languages appearing as a node in the FTG is also explicitly modelled. The transformations defined as activities in the PM are all modelled as rule-based graph transformations using AToMPM's transformation language (which was itself modelled explicitly). The FTG+PM language allows transformations to be defined as *automatic* or *manual*. Our framework allows user interventions in the MDE process, and provides means for creating artifacts using manual activities. The process model connects the transformations using control flow and data flow links. UML 2.0 activity diagrams were chosen as the language to describe the PM. This allows us to model the chaining of transformations as a process model and to build execution support for it. For execution, we map the process model to the native transformation scheduling language of AToMPM. The mapping takes into consideration whether a transformation is automatic or manual. In case of manual activities, the users can complete the task at hand and resume the execution of the process model which continues with the execution of the next scheduled transformation. The FTG+PM approach was applied to a concrete problem in the automotive domain: the power window case study.

As mentioned in Sect. 1, the goal of having a framework that allows us to thoroughly describe and automate model transformation chains is to give use the means to study and optimize such chains. As such we are currently developing the following:

– We currently use the power window case FTG+PM to study the notion of *intent* in model transformations. In our work in [29], the *intent* of a model transformation is defined as "a description of the goal behind the model transformation and the reason for using it". The FTG+PM model of the power window model transformation chain helped us to construct a transformation intent language. We are currently building a catalogue of model transformation intents (akin to design patterns in the OO world) and are formalising the properties of such intents. As mentioned in [30], the study of the formal properties of model transformations is in its infancy;
– As a result of our *transformation intent* work, we are now attaching *intent*-related annotations to the transformations described in the PM part of an FTG+PM model. Such annotations may serve to identify formal properties that should be proved for a model transformation. As transformation chaining is a form of relational composition, the formal composition of the properties of individual transformations in the chain is of great importance;
– Using the concrete power window case, we are also investigating the multi-paradigm modelling aspects of the FTG+PM [14]. We expect the study to help in identifying methodological and reusability concerns when developing model transformation chains for the automotive domain, that can hopefully be extrapolated to other domains.

Acknowledgements. Part of this work has been developed in the context of the NECSIS project, funded by the Automotive Partnership Canada.

References

1. Sendall, S., Kozaczynski, W.: Model Transformation – The Heart and Soul of Model-Driven Software Development. IEEE Software 20(5), 42–45 (2003)
2. Oldevik, J.: Transformation Composition Modelling Framework. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 108–114. Springer, Heidelberg (2005)
3. Vanhooff, B., Van Baelen, S., Hovsepyan, A., Joosen, W., Berbers, Y.: Towards a transformation chain modeling language. In: Vassiliadis, S., Wong, S., Hämäläinen, T.D. (eds.) SAMOS 2006. LNCS, vol. 4017, pp. 39–48. Springer, Heidelberg (2006)
4. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI – A Unified Transformation Infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
5. Heidenreich, F., Kopcsek, J., Aßmann, U.: Safe composition of transformations. Journal of Object Technology 10(7), 1–20 (2011), `http://dx.doi.org/10.5381/jot.2011.10.1.a7`

6. Wagelaar, D.: Blackbox composition of model transformations using domain-specific modelling languages. In: Proceedings of the First European Workshop on Composition of Model Transformations, pp. 15–20 (2006), http://doc.utwente.nl/66171/1/00000179.pdf
7. Seibel, A., Hebig, R., Neumann, S., Giese, H.: A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 19–39. Springer, Heidelberg (2012)
8. Kleppe, A.: MCC – A Model Transformation Environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer, Heidelberg (2006)
9. Mannadiar, R.: A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling. PhD thesis, McGill University (2012), http://msdl.cs.mcgill.ca/people/raphael/files/thesis.pdf
10. Giese, H., Levendovszky, T., Vangheluwe, H.: Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 252–262. Springer, Heidelberg (2007)
11. Dörr, H.: Efficient Graph Rewriting and Its Implementation. LNCS, vol. 922. Springer, Heidelberg (1995)
12. de Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Meta-Modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
13. Lúcio, L., Joachim, D., Vangheluwe, H.: An Overview of Model Transformations for a Simple Automotive Power Window. McGill University, Technical Report SOCS-TR-2012.2 (2012), http://msdl.cs.mcgill.ca/people/levi/30_publications/files/tech_report_mcgill_SOCS-TR-2012.2.pdf
14. Mustafiz, S., et al.: The FTG+PM framework for Multi-Paradigm modelling – An automotive case study. Paper for 6th International Workshop on Multi-Paradigm Modeling (2012), http://avalon.aut.bme.hu/mpm12/papers/paper17.pdf
15. Mosterman, P., Vangheluwe, H.: Computer Automated Multi-Paradigm Modeling – An Introduction. Simulation 80(9), 433–450 (2004)
16. Dorf, R.C.: Modern Control Systems, 12th edn. Pearson (2010)
17. Lúcio, L., et al.: The formalism transformation graph as a guide to model driven engineering. McGill University, Technical Report SOCS-TR-2012.1 (2012), http://msdl.cs.mcgill.ca/people/levi/30_publications/files/tech_report_mcgill_SOCS-TR-2012.1.pdf
18. Bottoni, P., Saporito, A.: Resource-based enactment and adaptation of workflows from activity diagrams. Electronic Communications of the EASST 18 (2009), http://journal.ub.tu-berlin.de/eceasst/article/view/233
19. Favre, J.-M.: Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In: Language Engineering for Model-Driven Software Development, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004), http://drops.dagstuhl.de/opus/volltexte/2005/13
20. Favre, J.-M.: Foundations of Model (Driven) (Reverse) Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon. In: Language Engineering for Model-Driven Software Development, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2005), http://drops.dagstuhl.de/opus/volltexte/2005/21

21. Rivera, J., et al.: Orchestrating ATL model transformations. Model Transformation with ATL, 34–46 (2009), `http://docatlanmod.emn.fr/MtATL2009Presentations/PreliminaryProceedings.pdf`
22. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: *trans*ML: A Family of Languages to Model Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 106–120. Springer, Heidelberg (2010)
23. Paige, R., et al.: The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), pp. 162–171. IEEE Computer Society (2009)
24. Aldazabal, A., et al.: Automated model driven development processes. In: ECMDA Workshop on Model Driven Tool and Process Integration 2008, pp. 43–54. Fraunhofer IRB Verlag (2008), `http://www.modelbus.org/modelbus/images/stories/docs/7AutomatedDevelopment.pdf`
25. Diaw, S., Lbath, R., Coulette, B.: Specification and Implementation of SPEM4MDE, a metamodel for MDE software processes. In: Proceedings of SEKE 2011, pp. 646–653. Knowledge Systems Institute Graduate School (2011), `http://www.ksi.edu/seke/Proceedings/seke11/173_Samba_DIAW.pdf`
26. Chou, S.-C.: A process modeling language consisting of high level UML-based diagrams and low level process language. Journal of Object Technology 1(4), 137–163 (2002), `http://dx.doi.org/10.5381/jot.2002.1.4.a3`
27. Bendraou, R., et al.: Definition of an executable SPEM 2.0. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007), pp. 390–397. IEEE Computer Society (2007)
28. Object Management Group: Software & Systems Process Engineering Metamodel Specification (SPEM) Version 2.0 formal specification – formal/2008-04-0 (2008), `http://www.omg.org/spec/SPEM/2.0/PDF/`
29. Amrani, M., et al.: Towards a Model Transformation Intent Catalog. In: Proceedings of the First Workshop on the Analysis of Model Transformation (AMT 2012), pp. 3–8. ACM (2012)
30. Amrani, M., et al.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012), pp. 921–928. IEEE Computer Society (2012)