The FTG+PM Framework for Multi-Paradigm Modelling: An Automotive Case Study

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

[†]School of Computer Science
McGill University, Canada

[‡]Department of Mathematics
and Computer Science
University of Antwerp, Belgium

[§]TERA-Labs
Karel De Grote
University College, Belgium

{sadaf,levi,hv}@cs.mcgill.ca, joachim.denil@kdg.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. We have earlier proposed a conceptual framework, FTG+PM, 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 activities in the MDE lifecycle such as requirements development, domain-specific design, verification, simulation, analysis, calibration, deployment, code generation, execution, etc. In this paper, we apply the FTG+PM approach to a case study of a power window in the automotive domain. We present a FTG+PM model for the automotive domain, and describe the MDE process we applied based on our experiences with the power window system.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.10 [**Software Engineering**]: Design—*methodologies*

1. INTRODUCTION

In recent times, model driven engineering (MDE) has been adopted in industrial projects in widely varying domains. The automotive industry in particular is faced with many challenges and opportunities.

Earlier [11], we proposed the FTG+PM framework for model-driven software development. It is intended to guide developers throughout the MDE lifecycle. The idea behind the FTG is similar to the Formalism Transformation Lattice for coupling different formalisms introduced by Vangheluwe in [18]. We go a step beyond multi-formalism modelling, and use the notion of multi-paradigm modelling [15] as the basis of our work. In multi-paradigm modelling, every aspect of a problem is modelled explicitly, at the right level(s) of abstraction, using the most appropriate formalism(s). This is enabled by metamodelling and model transformation.

The *Formalism Transformation Graph (FTG)* is a hypergraph with *languages* as nodes and *transformations* as edges. It charts the relationships among the multitude of languages and transformations used to develop systems within a domain. The *Process Model (PM)* precisely models the control and data flow between the transformation activities taking place throughout the software development lifecycle starting from requirements analysis and design, to verification, simulation, and deployment. We use a subset of the UML 2.0 activity diagram formalism for the PM. Our framework is supported by AToMPM [12], A Tool for Multi-Paradigm Modelling, for building metamodels, transformations, and for executing the FTG+PM model transformation chain.

In this paper, we focus on the application of our approach and demonstrate the capabilities of the FTG+PM through the design of an automated power window. The case study is inherently complex due to its heterogeneity and thus representative of industrial case studies. The model artifacts within the FTG+PM range from abstract requirements and analysis models, to more concrete design models, to models of source code. The discrete-time, continuous-time, discrete-event, and hybrid formalisms used in the FTG are appropriate to the levels of abstraction used at different stages of the modelling process. 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) [2] code.

This paper is organised as follows: Section 2 describes the application of FTG+PM to the power window case study. Section 3 discusses related work in this area and Section 4 draws some conclusions.

2. THE POWER WINDOW CASE STUDY

We apply our approach to a concrete problem in the automotive domain: the power window case study. The heterogeneity of components in such a system make it a suitable candidate for illustrating the MPM nature of the FTG+PM framework.

A power window is an electrically powered window and is present in the majority of the automobiles produced today. The basic controls of a power window include lifting and descending the window. Functionalities are added to improve the comfort and safety of the vehicle's passengers.

In Figure 1, we depict a condensed version of the FTG+PM model we have built for developing the Power Window software controller. The power window FTG+PM was built based on our experience with a concrete, AUTOSAR-based physical realisation of a power window.

The model-driven development of the power window controller includes several phases all of which are encompassed in the FTG+PM model (Fig. 1). Depending on the intent, it is possible to construct different PMs based on the same FTG and to traverse a particular path in the graph and apply a subset of the process for the purpose of simulation

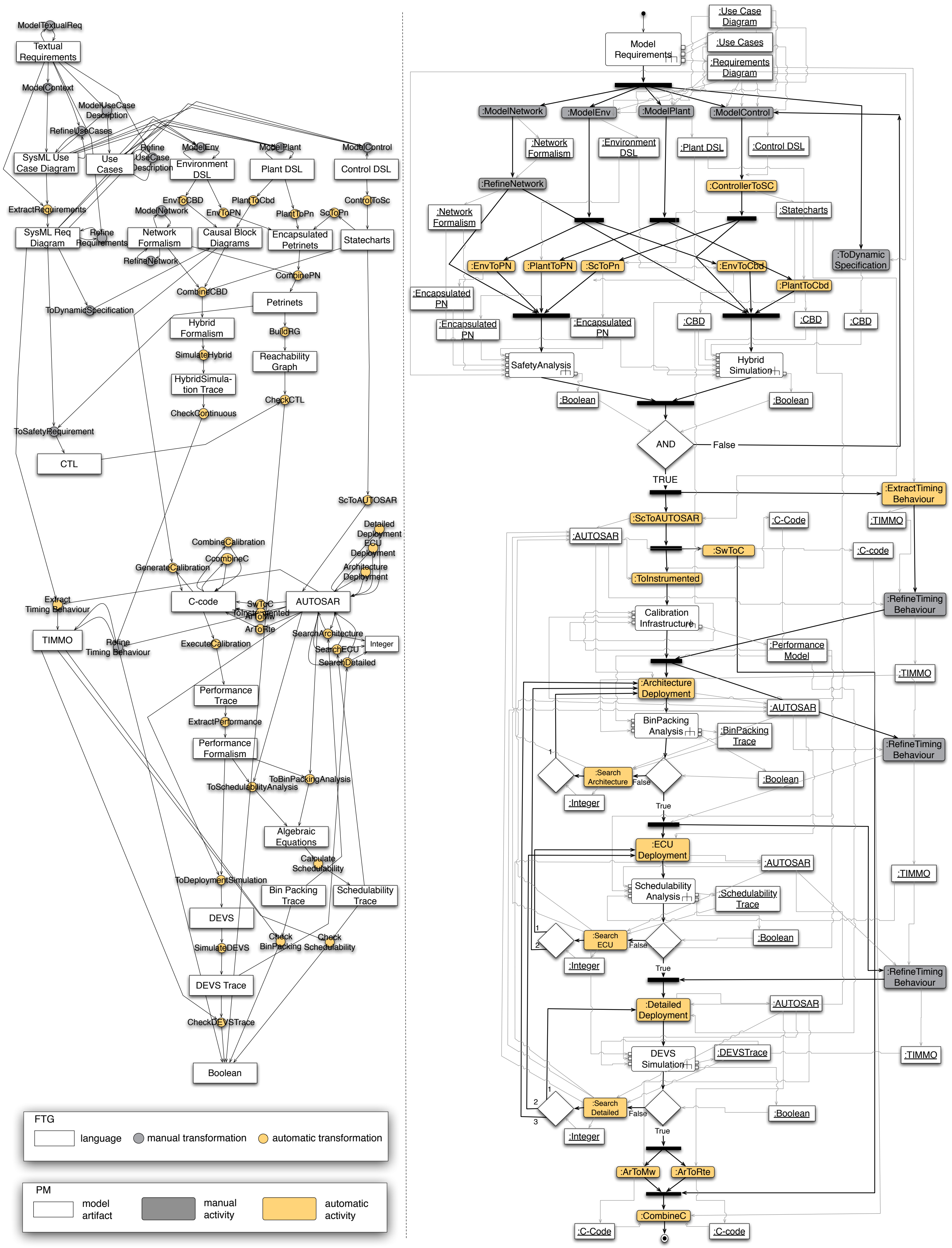

Figure 1: Power Window: FTG (left) and PM (right)

or verification (for instance). In the following sections, we briefly describe each activity and show some sample models and transformations. Due to space constraints, the metamodels of the languages and details of the transformations in the FTG+PM are not shown here. The interested reader can refer to [10] for further details.

We discuss the FTG+PM model of the power window based on the different dimensions and enablers of MPM presented in [15]: *levels of abstraction*, *formalisms*, *metamodelling*, and *model transformations*.

2.1 Levels of Abstraction

The power window controller is a complex, time-critical, safety-critical, hard real-time embedded system. When given the task to build the control system for a power window, two variables need to be considered: (1) the physical power window itself, which is composed of the glass window, the mechanical lifting mechanism, 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. This includes both human actors and other subsystems of the vehicle, e.g., the central locking system or the ignition system [15].

The level of abstraction is associated with the task to be accomplished and is determined by the perspective on the system, the problem at hand, and the background of the developer. At a high level of abstraction, the tasks in our MDE process are the development activities starting from requirements to code synthesis. The detailed tasks are declared as transformation definitions in the FTG and instantiated as activities in the PM.

The MDE process comprises several activities with models at different abstraction levels. Models at a higher level (starting from requirements models and DSMs) are refined until the executable model level (C source code) is reached. For each new modelling language, concrete syntax needs to be defined in addition to abstract syntax, tailored to the domain expert working on the specification of that model. Our approach integrates *multi-view modelling* by building distinct and separate models of the power window system to model different aspects of the system for different intentions. As an example, the domain-specific models are mapped to Petri nets with the intention of model checking and to CBDs[1] for simulation.

Systematically and automatically deriving models of different complexity significantly increases productivity as well as quality of models. In particular, our deployment activity (discussed later in this section) follows this principle.

The abstraction levels are based on the different MDE phases in the FTG+PM. We briefly discuss these phases here.

Requirements Engineering Before any design activities can start, requirements need to be formalised so they can be used by engineers. Starting from a *textual description* containing the features and constraints of the power window, a context diagram is modelled as a *SysML use case diagram*. The use cases are further refined and complemented with *use case descriptions*. Finally, the requirements are captured more formally in a *SysML requirements diagram*.

[1]Causal Block Diagrams (CBD) are a general-purpose formalism used for modelling causal, continuous-time systems, used in tools such as Simulink

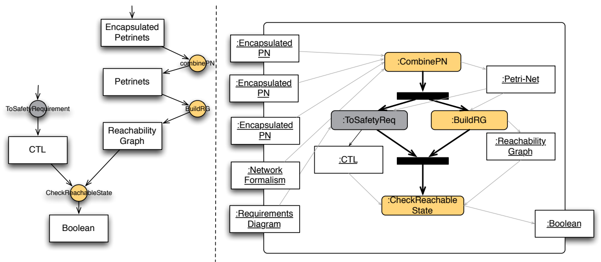

Figure 2: FTG+PM: Model Checking Slice

Domain-Specific Design 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*. Using the requirements models as a basis, we start the design activities by constructing domain-specific languages (DSLs) for the *Environment*, *Plant*, and *Controller*. A *Network* language is used to combine the three design languages and identify the interfaces between them.

Model Checking To ensure that there are no safety issues with the modelled control logic, formal verification can be carried out. The domain-specific models used for defining the plant, environment and the control logic are transformed to Petri nets where reachability properties are checked. It is then necessary to transform requirements to a property language (CTL, Computation Tree Logic in our case) so their satisfaction can be checked on the Petri nets. In Fig. 2, we present the safety analysis activity of the power window PM, along with the corresponding subset of the FTG. The FTG+PM makes causal relations between the different activities explicit.

Simulation The piecewise continuous behaviour of the up-and-downward movement of the window is simulated using a hybrid formalism. The hybrid model comprises the environment and plant models transformed into Causal Block Diagrams (CBD) and the controller in the Statecharts formalism.

Deployment After the software has been created and verified, it has to be deployed onto a hardware architecture, which 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. Deployment is the process of distributing these components over the hardware architecture and making other low-level choices such as scheduling. This can result in non-feasible solutions where the spatial and temporal requirements are violated. The deployment space can be searched for optimal solutions. In the power-window system, we only focus on the real-time behaviour, which is checked on three approximation levels. On these levels, bad solutions are pruned while good solutions can be explored further. Fig. 3 shows the actions involved in checking a single solution at the level of bin packing analysis. 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.

Calibration To build the performance model, we can also use generative MDE techniques. The plant model, environment model and instrumented source code are combined and executed in a Hardware-in-the-loop environment giving

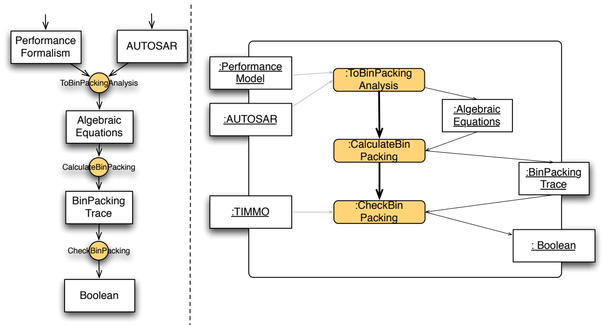

Figure 3: FTG+PM: Bin Packing Slice

back execution time measurements. These measurements are needed to calibrate a performance model that is used to guide the deployment space exploration.

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

2.2 Formalisms

We have used a multitude of languages to model the power window system at different levels of abstraction with different intentions. We have used a combination of UML modelling languages, domain-specific modelling languages, natural languages, and general purpose languages (GPLs).

- Requirements Development: specification of requirements using *textual requirements*, *SysML use case diagram*, *use cases*, and *SysML requirements diagram*
- Domain-Specific Design:
  - *Environment DSL* (to describe the interaction between actors and other subsystems)
  - *Plant DSL* (to describe the physical processes within the mechanical and electrical components)
  - *Control DSL* (to describe the logical operation of the hardware components)
  - *Network formalism* (to compose the DSMs by connections via ports)
  - *Statecharts* (to represent the reactive behaviour of the control DSL)
- Model Checking: DSLs are mapped to *encapsulated Petri nets* for carrying out safety analysis; reachability analysis uses *Petri nets* which generates a *reachability graph* model; safety and liveness properties are expressed in *CTL*
- Simulation: of continuous behaviour using a *hybrid simulation formalism* (based on causal block diagrams and Statecharts) and a *hybrid simulation trace language* (trace containing a time/signal value for the continuous part and a time/state value for the Statechart)
- Analysis: *TIMMO (TIMing MOdel)* [8] timing analysis, for the maximum end-to-end latencies of the application
- Calibration: calibration infrastructure for performance models in a custom *performance language* (timing properties of the software function w.r.t. a hardware type). The generation of the infrastructure is based on the *environment model*, *plant model*, and *instrumented source code (C GPL)*.
- Deployment:
  - software deployed onto a hardware architecture using the *AUTOSAR* middleware and metamodel
  - timing analysis using bin packing checks and schedulability analysis, based on *algebraic equations* and *bin packing and schedulability trace* languages (containing the results of the algebraic equations)
  - deployment simulation using *DEVS* (a modular and hierarchical formalism for modelling and simulating systems), generating a trace in the *DEVSTrace* language (containing the trace of the simulation), and a final result as a *boolean*
- Code Synthesis: generation of the application code, middleware and the AUTOSAR RTE in the *C GPL* language

2.3 Metamodelling

The languages are metamodelled using a base formalism, in our case class diagrams. The modelling environments are synthesized in AToMPM from their abstract and concrete syntax models. Due to space reasons, the metamodels are not shown here.

For the requirements languages, we use the standard SysML use case diagram and requirements diagram metamodels [1]. In case of the DSLs, the abstract syntax and the concrete syntax are constructed in AToMPM based on elements specified in the requirements. For Petri nets, the UML Petri net metamodel is used. For the encapsulated Petri nets, the Petri net metamodel is extended with ports and a *named* element class.

The meta-model of CBDs used is based on the work of Denckla and Mosterman [4]. It contains blocks, ports and relations between these ports. The abstract syntax used for Statecharts is based on the UML Statecharts metamodel, with semantics as defined by Harel [7]. The hybrid formalism combines the Statechart and CBD metamodels.

For the deployment part of this work, we use a subset of the AUTOSAR metamodel defined by the AUTOSAR consortium. The requirements language, TIMMO, extends the AUTOSAR metamodel with timing concepts defined in [8]. The algebraic equations language is based on simple mathematical formulas with float values, variables, equalities and algebraic operators. The algebraic traces gained from executing the algebraic equation contain a component name and float value representing either the load on the hardware component or the response time of the software function. The simulation model is based on the DEVS formalism. The DEVStrace language contains a time-stamp and action field. The boolean languages and integer languages just contain a single value with either a boolean or integer respectively.

The performance model also extends the AUTOSAR meta model with performance properties such as worst-case execution time, between a software function and a hardware type. This is gained by analysing the performance trace that contain the software function name with a float value (representing the execution time).

2.4 The Glue: Transformations

In this paper, we elaborate on a vertical slice of the FTG+PM. The transformations in the FTG take one of more models as input and produces one model as output. The models that are acceptable as inputs and outputs need to conform to the metamodels described in Sections 2.2 and 2.3. Between square brackets, we classify the transformations according to [13].

ModelContext derives a SysML use case diagram (Fig 4) based on the textual requirements. These transformations are usually done manually by requirements engineers. Some automatic transformations can be used to populate the use

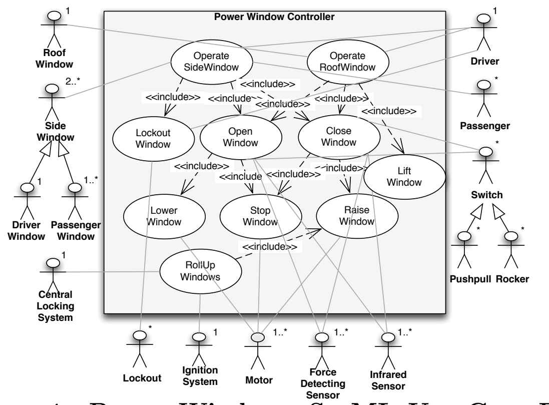

Figure 4: Power Window: SysML Use Case Diagram

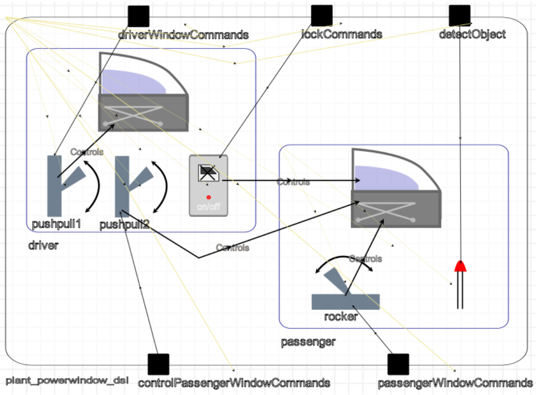

Figure 5: Power Window: Plant DSL Model

case diagram and requirements diagram. [exogenous, horizontal]

RefineUseCases is a transformation between models expressed as SysML use case diagrams (Fig 4). The refinement takes into consideration inputs and clarifications from the client. [endogenous, horizontal]

ExtractRequirements takes a SysML use case diagram and produces a SysML requirements diagram (not shown due to space constraints) which is refined with extra functional requirements and constraints based on the initial textual requirements. [exogenous, vertical]

ModelPlant maps the requirements (inputs: SysML requirements diagram, use cases) to a DSL (output: Fig. 5) to represent the *plant* domain. [exogenous, vertical]

PlantToPN takes a plant DSL model (Fig. 6) and transforms that to an encapsulated Petri net[2] [exogenous, vertical]

CombinePN takes three encapsulated Petri nets (not shown here) derived from the environment, plant and control domain-specific models, as well as a network model that specifies how the three communicate. The transformation outputs 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. [exogenous, horizontal]

ToSafetyReq takes as input a model of the safety requirements, as well as the combined Petri net model representing the behaviour of the whole system, and outputs a set of CTL formulas encoding the requirements. [exogenous, vertical]

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

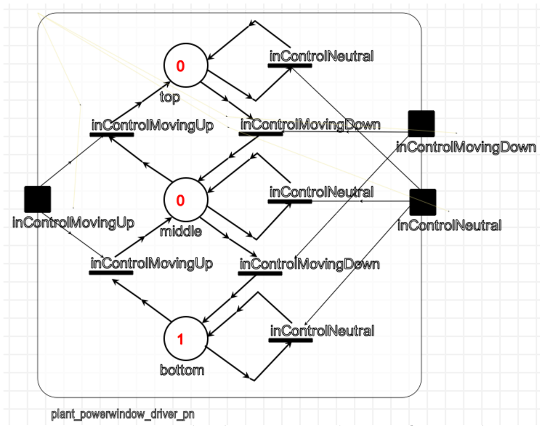

Figure 6: Power Window: Petri Net for Driver Window

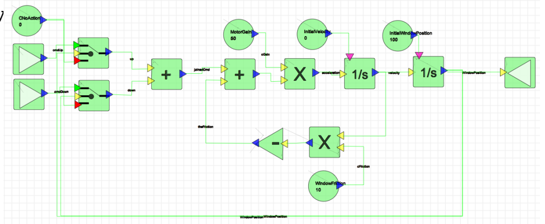

Figure 7: Power Window: Causal Block Diagram (CBD) for Plant Model

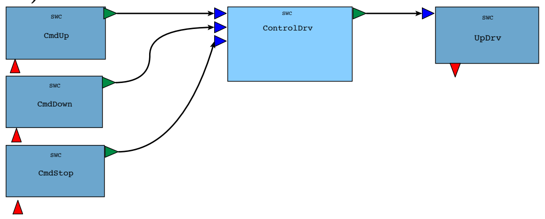

Figure 8: AUTOSAR Software Component Model

vertical]

Build RG takes the derived *Petri net* and generates a *reachability graph. Note that* ToSafetyReq *and* BuildRG *should be executed in parallel.* [exogenous, vertical]

CheckReachableState takes the CTL formulas and outputs *true* if the reachability analysis is successful. [exogenous, vertical]

PlantToCBD transforms the plant DSL to a causal block diagram (Fig. 7). [exogenous, vertical]

CombineCBD takes the CBDs mapped from the DSLs and the Statechart model, and composes them to produce a hybrid simulation model. [endogenous, horizontal]

SimulateHybrid uses the hybrid simulation model as the source and produces a mixed continuous - discrete trace model. [exogenous, vertical]

CheckContinuous takes the continous trace model, ensures that it conforms to the initial specified constraints, and outputs a boolean value depending on the success of the analysis. [exogenous, vertical]

ScToAUTOSAR encapsulates the given Statechart in an AUTOSAR-compliant model (Fig 8). A single software component is used for the Statechart while for each input signal to the Statechart, a sensor-actuator component is created. [exogenous, vertical]

SwToC generates AUTOSAR-compliant C code of the application from the input AUTOSAR model. [exogenous, synthesis]

ToInstrumented generates instrumented C code of the application from the AUTOSAR model. This can be used to create a performance model of the application used during deployment. [exogenous, synthesis]

Architecture Deployment adds information on the mapping of software to hardware components to the AUTOSAR model. Backtracking may take place during the exploration of the deployment space. [endogenous, vertical]

ToBinPackingAnalysis creates an output algebraic equation from the AUTOSAR model and the performance model to check the load of the hardware component. [endogenous, vertical]

CalculateBinPacking executes the algebraic equations created by the *ToBinPackingAnalysis* transformation. [exogenous, horizontal]

CheckBinPacking checks the result of the execution of the bin packing check with the RMA threshold defined by Liu and Layland [9]. [exogenous, horizontal]

SearchArchitecture decides how the exploration should proceed in case the schedulability test failed. [exogenous, horizontal]

ArToMW generates the middleware code for a single control unit from the deployed AUTOSAR model. [exogenous, synthesis]

ArToRTE generates the run-time environment code for a single control unit from the deployed AUTOSAR model. [exogenous, synthesis]

CombineC combines the application software code, the run-time environment code and the generated middleware code so it can be compiled. [endogenous, horizontal]

3. RELATED WORK

Research has been carried out in both academia and industry on the model-driven engineering of automotive cyber-physical systems [6, 19, 5]. [3] presents an MDE framework based on SysWeaver for the development of AUTOSAR-compliant automotive systems. Typical design methods used in domains such as automotive and aerospace follow the V model for software development [14, 16]. Prabhu and Mosterman [17] illustrates the model-based design of an embedded control systems using the power window case study. They cover the MDE process starting from behavioural modelling to code generation, and focus on the use of an integrated tool suite for MDE. Many links exist with research in process modelling. These are not described due to space constraints.

4. CONCLUSION

We have applied the FTG+PM framework to 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. This encompasses the various phases of model-driven development of the power window. As part of each phase, we defined the appropriate formalism(s) and the relations between them. The process model was used to guide the development of the power window controller.

The FTG and PM we have presented 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 believe that our experiences can be useful for others working on the development of systems in the automotive domain. The power window FTG+PM is a skeleton which can be extended, refined, or adapted for various techniques and technology, for example feature modelling for software product lines. We intend to study higher order characteristics of transformation chains, and use the FTG+PM to automatically reason about the properties of chains of model transformations. We further plan to work on integrating the model-based testing phases in our framework and applying it to our case study.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

[1] OMG Systems Modeling Language, OMG document number formal/2012-06-01 (OMG SysML version 1.3), available from www.omg.org, 2012.

[2] AUTOSAR. `http://www.autosar.org`, 2010.

[3] 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.

[4] B. Denckla and P. J. Mosterman. Formalizing Causal Block Diagrams for Modeling a Class of Hybrid Dynamic Systems. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 4193–4198, 2005.

[5] J. Friedman and J. Ghidella. Using model-based design for automotive systems engineering - requirements analysis of the power window example. *SAE*, 2006.

[6] 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.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[8] R. Johansson, P. Frey, J. Jonsson, J. Nordlander, R. M. Pathan, N. Feiertag, M. Schlager, H. Espinoza, K. Richter, S. Kuntz, H. Lonn, R. Kolgari, and H. Blom. TIMMO Timing Model TADL: Timing Augmented Description Language. Technical report, 2009.

[9] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[10] L. Lucio, J. Denil, and H. Vangheluwe. An overview of model transformations for a simple automotive power window. Technical Report SOCS-TR-2012.2, McGill University, January 2012.

[11] L. Lucio, S. Mustafiz, J. Denil, B. Meyers, and H. Vangheluwe. The formalism transformation graph as a guide to model driven engineering. Technical Report SOCS-TR-2012.1, McGill University, March 2012.

[12] R. Mannadiar. *A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling.* PhD thesis, McGill University, June 2012.

[13] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, Mar. 2006.

[14] P. Mosterman, J. Sztipanovits, and S. Engell. Computer-automated multiparadigm modeling in control systems technology. *IEEE TCST*, 12(2):223 – 234, 2004.

[15] P. J. Mosterman and H. Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation*, 80(9):433–450, 2004.

[16] K. Muller-Glaser, G. Frick, E. Sax, and M. Kuhl. Multiparadigm modeling in embedded systems design. *IEEE TCST*, 12(2):279 – 292, march 2004.

[17] S. Prabhu and P. Mosterman. Model-based design of a power window system: Modeling, simulation and validation. In *IMAC-XXII*, 2004.

[18] H. Vangheluwe and G. C. Vansteenkiste. A multi-paradigm modeling and simulation methodology: formalisms and languages. In *ESS'96*, pages 168–172, 1996.

[19] S. Wang. Model transformation for high-integrity software development in derivative vehicle control system design. In *HASE*, pages 227–234, 2007.