An Overview of Model Transformations for a Simple Automotive Power Window

[†]Levi Lucio, [‡]Joachim Denil, [†,‡]Hans Vangheluwe
[†]McGill University, [‡]University of Antwerp

15 January 2012

Abstract

The goal of this report is to illustrate the *Model-Based Development* of a simple automotive control system. The *Power Window* example was chosen as one the hand, it is simple enough to not require extensive automotive and control theory background, and on the other hand, it is complex enough to be representative for Model-Based Development in the automotive domain. This complexity mainly stems from the use of models in different formalisms. Models in different formalisms are composed as well as transformed. Furthermore, this example was chosen as we already have some concrete experience with this case, having built a full hardware realization of it using Model-Based techniques.

This example, and this document, are meant as a *starting point* for further discussion among NECSIS partners (and in particular, with GM, to discuss correctness and completeness).

It is hoped that after sufficient refinement, this example can be used as a *common case study* by NECSIS partners, each focusing on different aspects (such as safety or variability) and techniques (simulation, code synthesis, model checking of models and transformations). As such it can be used as a *benchmark* for individual researchers to test their own approaches and techniques. If a common representation is used, exchange of models will be possible and different techniques developed by the partners can be combined.

The main focus of this document, developed within the NECSIS Theme on "analysis and transformation" is *transformation*. Different types of transformations are identified and described by means of their typical input-models and expected output models.

Note that though we have already implemented some of these transformations, they are left out of this document on purpose not to bias other researchers in their design of transformations satisfying the requirements. A first attempt is made to categorise the properties of the transformations. Also, the link between the different transformations is depicted in a model *transformation graph*.

This document is a first version of the transformations required for developing the software to control a Power Window. We are currently working on: a better structuring of the transformation chains presented here in the form of a formalised *transformation graph*; a categorisation and classification of the of model transformations that can help us identifying their properties.

1 Introduction and Problem Statement

In this document we will introduce a study of the application of Model Driven Engineering (MDE) techniques to the development of the software necessary for the operation of a power window in an automotive vehicle. In particular, the main purpose of this study is to arrive to a set of model transformations

Our first goal throughout this document is to present an MDE example approach to the development of the software controller for the power window. Using MDE implies: 1) defining or reusing a set of formalisms for the specification of models; 2) defining or reusing a set of transformations between those formalisms. Using these artifacts it is possible to define a chain of development (or methodology) where models of software expressed in languages familiar to humans are submitted to a number of transformations into appropriate formalisms such that eventually executable code that can be run on a machine is reached.

2 Background Concepts

MDE encompasses both a set of tools and a loose methodological approach to the development of software. The claim behind Model Driven Engineering is that by building and using abstractions of the processes the software

engineer is trying to automate, the produced software will be of better quality than by using a general purpose programming languages (GPL). The reasoning behing this claim is that abstractions of concepts and of processes manipulating those concepts are easier to understand, verify and simulate than computer programs. The reason for that is that those abstractions are close to the domain being addressed by the engineers, whereas general GPLs are built essencially to manipulate computer architecture concepts. Of course, the reasoning holds when the used abstractions are suitable to describe the addressed domain.

2.1 Models and Metamodels

The central artifact in MDE is the *model*. A model in the computing world is a simplification of a process one wishes to capture or automate. The simplification is such that it does not take into account details that can be overseen at a given stage of the engineering cycle. The purpose is to focus on the relevant concepts at hand – much as for example a plaster model of a car studying aerodynamicity will not take into account the real materials a car is made of.

In the computing world a *model* is defined by using a given language. Coming back to the car analogy, if an engineer wishes to have a computational model of a car for three dimensional visualisation, a language such as the one defined by a Computer Assisted Design (CAD) tool will be necessary to express a particular car design. In the computing world 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 (also called *formalism*) that may have many *model* instantiations, much as in a CAD tool many different car designs can be described.

2.2 Domain Specific Modeling

This leads us to the notion of *Domain Specific Modeling*. Domain Specific Modeling formalizes the fact that certain languages, or classes of languages – called Domain Specific Languages (DSLs) – are appropriate to describe models in certain domains. A famous white paper on the subject from Metacase^{TM} [11] presents annecdotal evidence that DSLs can boost productivity up to 10 times, based on experiences with developing operating systems for cell phones for Nokia^{TM} and Lucent^{TM}. Following these encouraging first results the scientific community is currently investing on the research of DSLs and environments for the construction of such languages. The result of this research has materialized in formalisms and tools such as EMF and GMF [12], AToM^3 [5] or Microsoft's^{TM} DSL Tools [4].

2.3 Multi-Paradigm Modeling

Multi-Paradigm Modeling (MPM), as introduced by Mosterman and Vangheluwe in [13], is a perspective on software development that advocates not only that models should be built at the right level of abstraction regarding their purpose, 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 modeling as an activity that spans different models, or 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 inbetween formalisms to specialized machinery.

To make this idea more concrete, one may think of a UML statechart model representing the abstract behavior of a software system being converted into a Java model for execution on a given platform; or of the same statechart being transformed into a formalism that is amenable for verification. Another possible advantage of this perspective on software development is the fact that toolsets for implementing a particular software development methodology become flexible. This is due to the fact that formalisms and transformations may be potentially plugged in and out of a development toolset given their explicit representation.

The idea of Multi-Paradigm Modeling is close to the idea of Model Driven Engineering (MDE): in MPM the emphasis is mainly on the fact that several modeling paradigms are employed in modeling; MDE is rather focused on proposing a methodology where a set of model transformations are chained in order to pass from a set of requirement for a system to a piece of running software on a given platform.

2.4 Model Transformations

The missing piece in this set of concepts are *Model Tranformations*. Model transformations allow passing relevant information from one modeling formalism to another and are, according to Sendall and Kozaczynski [17] the "heart and soul of model-driven software development". Model transformations have been under study from a theoretical point a view for a number of years (see e.g. the work of Ehrig et al [9]), but only recently have become a first

class citizen in the the software development world. Their need came naturally with the fact that MDE started to be used professionally in some software development environments, e.g. software for mobile phones or software for the automotive industry. Implementations for transformation languages such as ATL [2] or QVT [8] have been developed in the last few years and provide stable platforms for writing and executing model transformations.

Model Transformations can have multiple uses in MDE: for example, if it becomes necessary to transform a UML statechart into code a model transformation can be seen as a *compiler*; also, a transformation to translate the statechart into a formalism amenable to verification by some existing tool may be seen as a *translator*. These transformations clearly exist in traditional software development, although in an implicit fashion. Being that in an MDE setting model transformations are responsible for translating models from one formalism into another, it becomes important for the quality of the whole software development process that those transformations are correct.

Model transformations are pieces of software. The verification of software is typically achieved by proving that a given program has certain formal properties that insure a certain level of correctness regarding that program's specification. The approaches in the literature to the problem of verifying model transformation can be divided in two main groups: some authors such as Akehurst [1], Narayanan and Karsai [15] or Lúcio and Barroca [10] are interested in proving that certain relations hold between the grammar of the input model and the grammar of the output models. In order to clarify this approach consider one want to prove an automatic English to French translator works properly. It might be interesting to prove that all english sentences composed of a *verb* and of a *predicate* will be translated into a french sentence composed equally of a *verb* and a *predicate*. Such a vision of correctness of a model transformation can be seen as *syntactic* since the proof does not explicitly take into consideration the meaning of the input and output models, but rather their structure. The second group of approaches is explicitly concerned with proving that parts of the meaning, or *semantics*, of input models are given the correct semantics in the output models. Again, to give a concrete example, one may wish to show that if in a statechart model a certain kind of user is always be able to execute a given activity, then in the Java code generated from that statechart the transformed user will still be able to execute the transformed activity. Authors such as Varró and Pataricza [20] or Narayanan et al [14] have proposed techniques to deal with semantics preservation when model transformations are applied.

3 Engineering the Power Window Software

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 increase 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.

When given the task to build the control system for a power window, a software engineer will take several 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 [13]. According to control theory [7], 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*.

In the next few sections we will go through a possible set of model driven engineering activities when building the software *controller* for the power window system. We will start by the modeling activities which involve developing domain specific languages for defining the *controller*, the *plant* and the environment. We will then carry on to explain how models for verification, simulation and finally code generation can be achieved from the three initial models.

Note that in what follows we do not mean to be prescriptive or complete regarding the activities we are going to describe or the adopted engineering methodology. Our purpose is to provide a reasonable example of MDE in the automotive software development context, that illustrates the usage and power of state of the art MDE artifacts and tools.

3.1 Modeling Activities

As mentioned before, we consider that during the development of a software controller for a power window it is necessary to take into consideration both the description of the physical hardware itself – the *plant* – as well as the description of the environment interacting with the power window. While the fact that we need to take the plant into consideration when building a controller is self explanatory, the *environment* requires further analysis. Several publicly available documents such as [18, 19] describe safety measures associated with the operation of a power window (in the generality of the term, taking into consideration also space partition panels and sun roofs). In these documents several situations of interactions of the power windows with humans (and an external environment) are mentioned. The largest concern is clearly the situation where where somebody becomes unintentionally physically caught by a closing power window. Some power window systems include automatic reversal systems which detect if an object is blocking the windows path and automatically stops the window's movement. Other systems do not include the automatic stopping and backing system and a number of other strategies are put into place such that accidents where somebody becomes caught by the power windows (typically children) do not happen. In order to automatically verify the power window and the car in general has correctly implemented those strategies we will take into consideration the actors in the environment (e.g. adults, children) and model their interactions with the power window and relevant parts of the car.

We will now introduce three domain specific languages that will allow us to describe power window *plants*, *controllers* and *environments*. Our idea behind splitting these three artifacts involved in the development of a power window is the fact that they describe somewhat orthogonal concerns that can be replaced in a modular fashion during the development of the same, or multiple power window controllers. Followed by each domain specific language we will describe one of its *models* both as an example and also as a means of providing input for subsequent stages of engineering of the power window controller.

Note that in the text that follows all the DSLs' grammars are described using metamodels expressed as class diagrams. Metamodels essentially identify and define the components of a language (described as *classes*), and the relations between those components (described as *associations* between classes). Note also that, because we have taken a modular approach to the development of the PowerWindow software, our formalisms allow encapsulation in the sense that the models of the different languages will be contained inside *boundaries*. These *boundaries* contain *ports* that are linked to the objects inside. In order to compose models describing different aspects of the PowerWindow, in what follows we will also describe a *Network* language that connects *ports* of multiple formalisms together.

3.1.1 Power Window Description Language

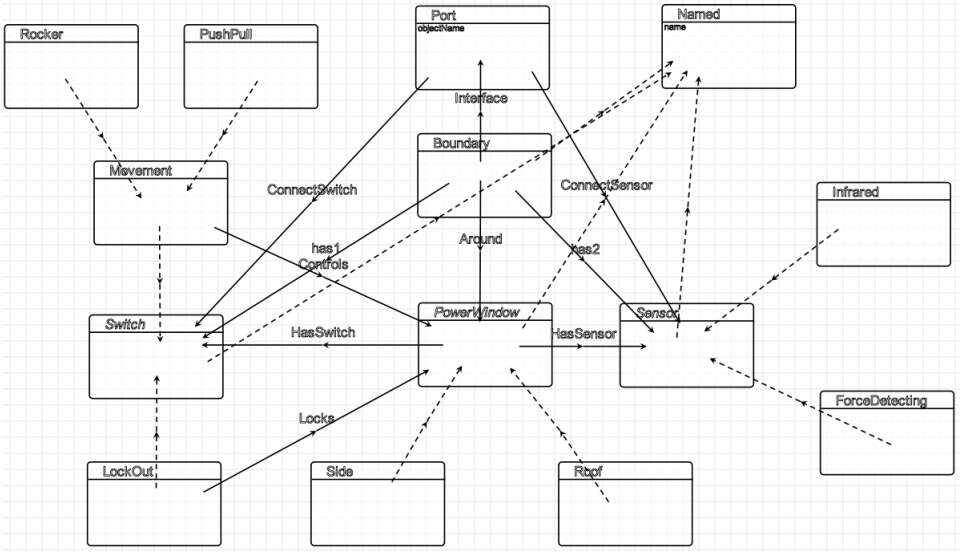

Figure 1: Power Window Description Language Metamodel

In figure 1 we present the metamodel for the a language allowing to describe the *plant* for the controller we wish to

develop. In the power window case this amounts to a language allowing the description of the hardware configuration for a given power window device.

Before we start the description of the Power Window Description Language (PWDL) metamodel, take note that classes in a metamodel can be *regular* or *abstract*. Abstract classes have their name in italicised font and, unlike regular classes, cannot be instantiated. On the other hand relations can be of two kinds: *association*, noted with a full arrow, or *inheritance*, noted with a dashed arrow. Note that both association and inheritance are directed relations and also that associations can have *containment* semantics. Containment semantics means that the existence of an instance of the class at the association's end implies the existence of an instance of the class at the association's origin.

Coming back to the PWDL, the main class of the language is the *PowerWindow* class, which is abstract and is specialized as a *Side* window or a *roof* window. A Powerwindow includes a set of switches which can be of two kinds: *Lockout* switches allow removing control from other PowerWindows in the car (as specified by the *controls* association); *Rocker* or *PushPull* switches which allow controlling window movement. *Rocker* and *PushPull* switches have different physical characteristics: while Rocker switches are used to activate window movement by being acted upon on the horizontal axis, pushpull switches need to be acted upon on the vertical axis by being pushed down or pulled up. Finally, a Powerwindow may also have sensors for detecting if an object is blocking the window from going up. These sensors may be of two types: *Infrared* or *ForceDetecting*. *Infrared* sensors detect an object when a light beam is crossed; *ForceDetecting* sensors make use of the fact that if an object is being pushed up by the window then more current will be drawn by the electrical engine.

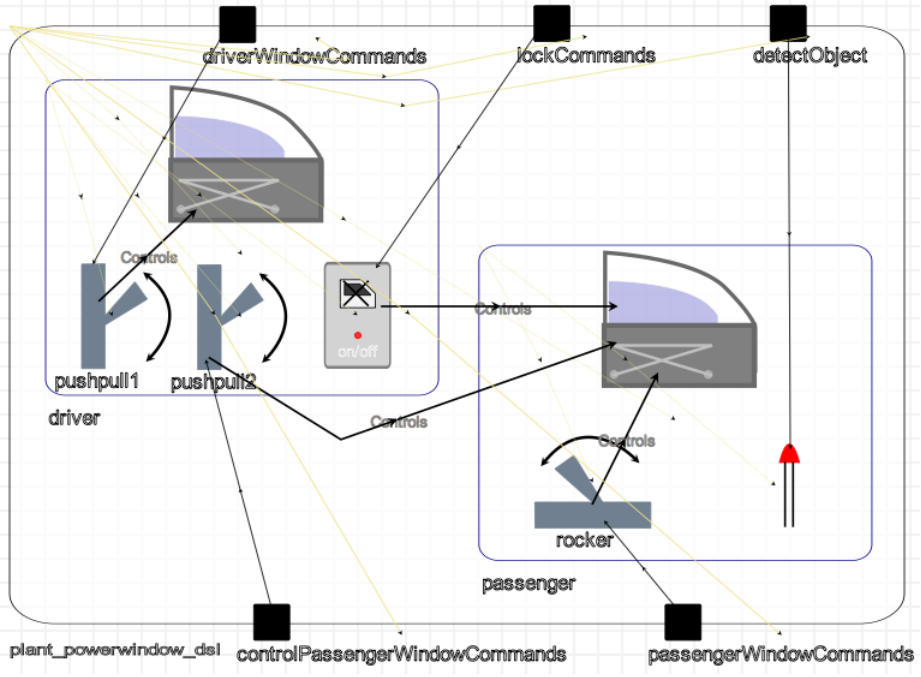

Figure 2: Example Model in the Power Window Description Language

In figure 2 we present a model in the PWDL, where a configuration of two powerwindows of an automobile is described. The model includes a *driver* and a *passenger* powerwindows, 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 lifting when an object obstructs its path.

3.1.2 Environment Description Language

The *Environment* description language, which metamodel is described in figure 3, allows describing interactions of the outside world with the PowerWindow. The language is inspired from the work of Dhaussy [6] and has been applied to the verification of critical systems, e.g. in military aviation [16].

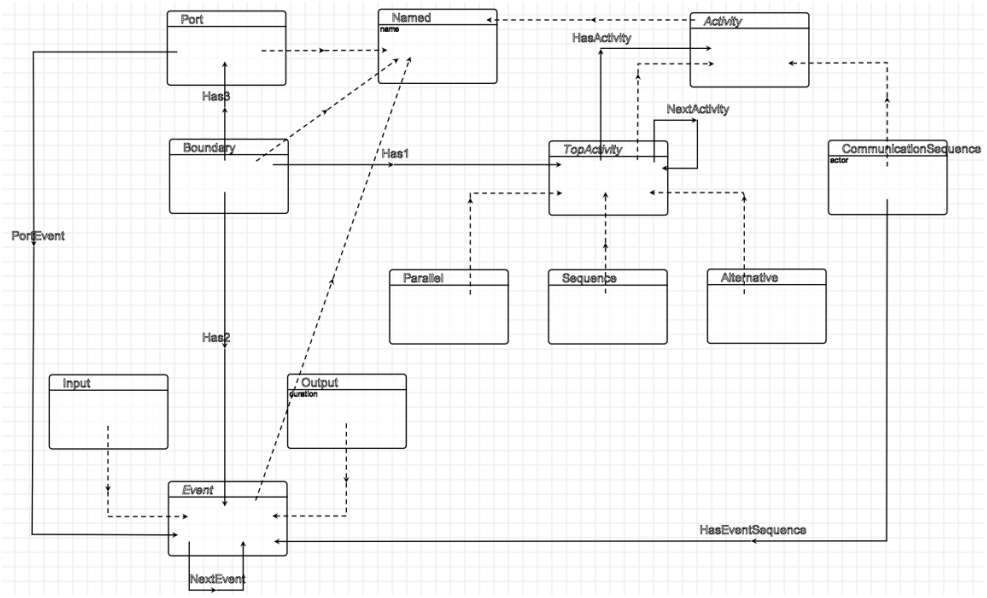

Figure 3: Environment Description Language Metamodel

Every model in the environment description language has a *top activity*, which can be of type *parallel*, *sequence* or *alternative*. A top activity may contain other activities, in an arbitrarily deep hierarchy. As the names indicate, activities which are executed in parallel will occur simultaneously, sequential activities will occur one after the other in a predefined fashion, and, from a set of alternative activities, one is chosen non-deterministically to be executed. The basic activity is the *communication sequence* which is a sequence of events being exchanged with another system. Events may be of type *output*, meaning they are sent towards the outside system, or of type *input*, meaning they are expected from the outside system. An event has an amount of time in seconds associated to it, which is the amount of time the event will take to complete.

In figure 4 we present a model described in the environment description language. The model presents an example of an interesting interaction with the powerwindow hardware described in figure 2 where the driver and the passenger both issue a sequence of commands in parallel. The parallel block represented by the red square includes two command sequences that have a sequence of output events. Each graphical representation of an output event includes on the right of the box the amount of time the event will take until completion. In particular, the *stickhead* command means the passenger has blocked the window rolling up by blocking it with some object. Note that an environment could contain other actors, even other systems. Interesting examples would be the ignition system which could interact with the powerwindow by disabling window action when key is off the ignition, or the speedometer which could force rolling up the windows once a given speed would be reached.

3.1.3 Control Description Language

The control description language defined as in the metamodel in figure 5 allows describing the operation of the plant according to the plant's current state as well as to the inputs received from the environment. The language is inspired from UML statecharts and can be seen as a language for describing finite state automatons.

In figure 6 a model for controlling a powerwindow with object detection can be observed. The control logic states that the window can be either in neutral mode (with the electrical engine stopped), moving up or moving down. The control logic will change state if a new control command will be issued by one of the buttons attached to the window. The model includes dealing with stopping window action if an object is detected blocking window lifting. In this case the window control logic goes into an emergency state and then into the neutral state which stops window movement (review this part because the neutral state has to synchronize with the plant).

In figures 7 and 8 we present additional models in the control description language required to describe the control aspects of a powerwindow without obstacle detection, as well as of a lockout button.

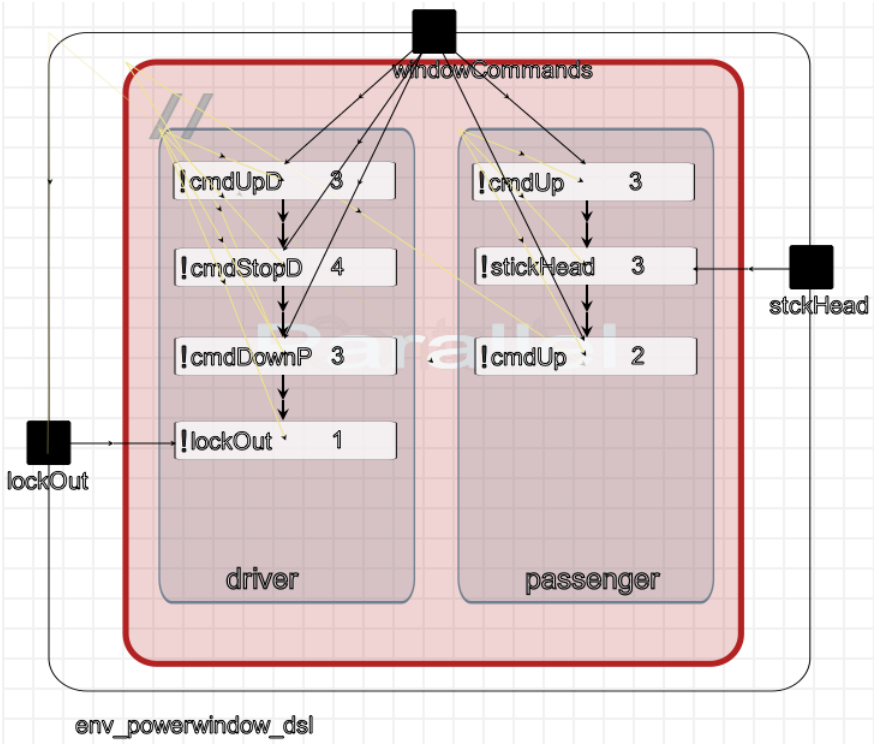

Figure 4: Example Model in the Environment Description Language

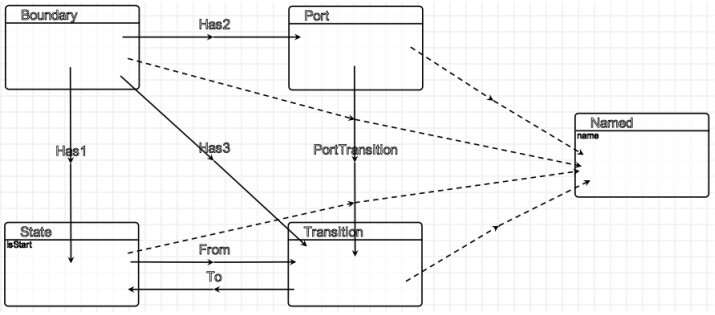

Figure 5: Control Description Language Metamodel

3.1.4 Network Description Language

It becomes now necessary to compose the *plant*, *environment* and *control* models presented in figures 2, 4, 6, 7 and 8. In figure 9 we present the metamodel of a Network Description language used for this task. The idea behind this language is very simple: components can be connected to other components, where a component is seen as a black box and connections are made by linking components' ports. Each of the formalisms introduced in sections 3.1.1, 3.1.2 and 3.1.3 has the notion of *ports*, represented in concrete syntax by the black squares over the boundaries of each model.

In figure 10 the network model for our powerwindow example is presented. Note that the internal detail of each of the models is abstracted and only the connections between the ports are visible.

3.2 Verification Activities

We will now describe the generation of a Petri Net from the DSL models described in section 3.1. The Petri Net formalism is an automaton like formalism involving places and transitions (resembling the UML states and transitions) but also with the capability of describing concurrency. Tokens distributed by places allow describing

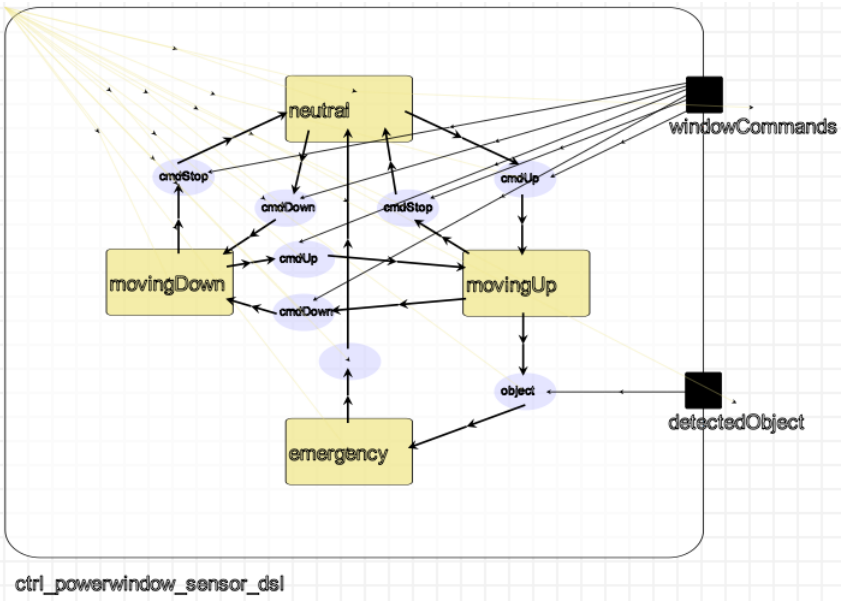

Figure 6: Window with Obstacle Detection Control Model

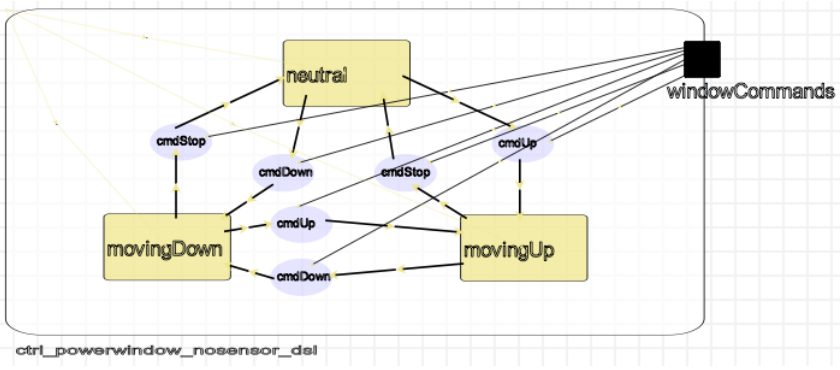

Figure 7: Window with No Obstacle Detection Control Model

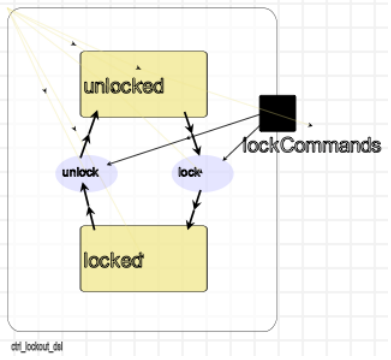

Figure 8: LockOut Control Model

that certain resources are distributed in the system and the non-deterministic firing of transitions simulates the consumption of resources in places and production of new resources in other places.

The goal of generating a Petri Net from the DSL models is to build an artifact that can be used for the exhaustive exploration of functional scenarios of operating the powerwindow. Many model checking tools exist that will take

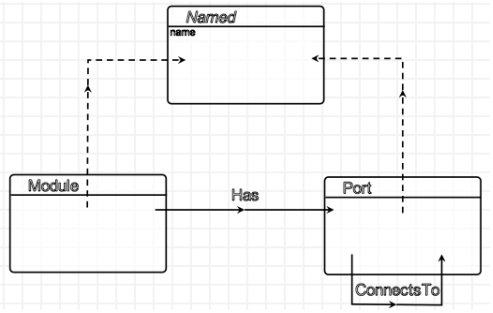

Figure 9: Network Description Language Metamodel

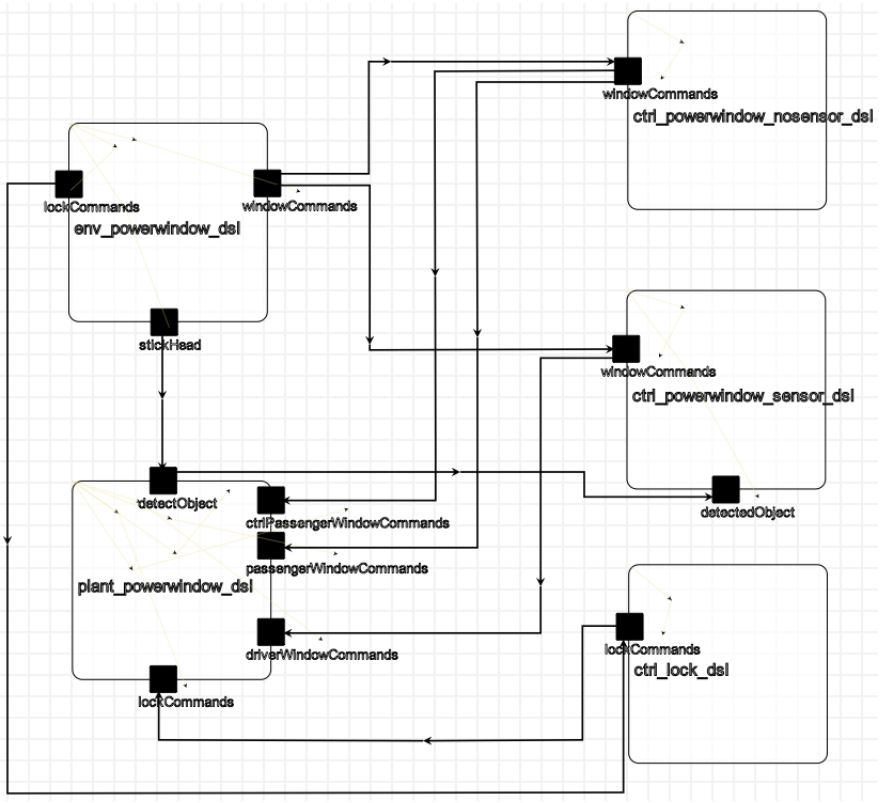

Figure 10: Network Model for the Powerwindow

as input a Petri nets model and a property and will decide if the property is true or not in the model. This Petri net may be used to verify certain properties hold in our powerwindow definitions. For example, it may be important to show that if the driver is commanding the passenger window to go up or down, then the passenger cannot operate his/her window; or, that when an obstacle blocks a sensor-equipped powerwindow when going up, the window will stop. In this section we will describe how the Petri net representing the behavior of the composition of the DSL models presented in section 3.1 is obtained by using a set of transformations. We will however not concern ourselves with the properties to be proved about the Petri net representation of the system, which would be the target of a different kind of study than the one presented in the present document.

In what follows we will start by presenting the transformation of each of the DSL models presented in section 3.1 into Petri nets. We will also transform the network model into a specialized network model that will be used in the composition of the Petri nets obtained from the DSL models. Notice that Petri nets are a discrete formalism, as opposed to the causal block diagram formalism presented in section 3.3 which is a formalism capable of representing continuous behavior. The discrete nature of Petri nets comes from the fact that every state of the execution of a Petri net model can be identified as a set of tokens occurring in a set of places.

Notice that in the following sections we will use a particular kind of Petri nets which we call *modular Petri nets*. Modular Petri nets are an extension of the Petri Net formalism where regular Petri nets are encapsulated by *boundaries*. Those boundaries expose *ports* to the outside of the system. Petri net transitions inside the boundaries connect to the module's ports, which means they can be synchronized with other modules via a *network* model.

3.2.1 Transformation of the Environment Model into Petri Nets

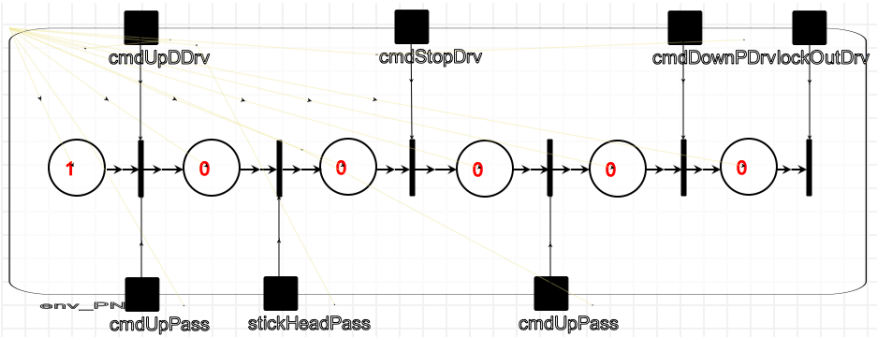

Figure 11: Transformed Environment in Encapsulated Petri Nets

In figure 11 we depict the result from the transformation of the environment DSL model presented in figure 4 into modular Petri nets. The transformation recursively treats all the *parallel*, *sequence*, *alternative* and *communication sequence* activities in the environment model and builds the necessary modular Petri net for them. In the modular Petri net in figure 11 we can observe that the two communication sequences from the environment model in figure 4 have been merged into one single sequence of transitions. This is so because the two activities occur during the same timeline, where each relevant discrete moment is represented by a transition. The transformation from the environment DSL into a Petri net will compute how many discrete moments are required and will generate a corresponding amount of Petri net transitions. The ordering of the events depends on the amount of time each of them requires to be completed. The Petri net transitions are synchronized with ports of the module in order to output the events to other components. Some of the events can happen simultaneously (because their time distance from the start of the activity is the same), which means the same Petri net transition connects to two ports.

The translations of the *alternative* and *sequential* blocks are less complex than the ones of the *parallel* block because the timelines for each of those blocks do not need to be composed, although they need to be assembled together such that one or the other is chosen (alternative), or they are sequentially executed (sequence). A final observation on the transformation of environment models into modular Petri nets is the fact *input* events coming from the outside of the components are treated by synchronizing the Petri net's transitions with them. This is done is such a way that a transition of the component can only fire when receiving an input from an external component.

3.2.2 Transformation of the Plant Model into Petri Nets

Of the two windows in figure 13, the *driver* window does not include an obstacle detection sensor, while *passenger* window does include an infrared sensor. Two modular Petri nets are generated from the Plant DSL model: in figure 12 a discrete behavior of a powerwindow without an obstacle detecting sensor can be observed. During operation the window can either be at the *bottom* of the frame (meaning the window is completely open), somewhere in the *middle* of the frame (meaning the window is partially open), or at the top of the frame (meaning the window is closed). The behavior of the plant is dependent on the behavior of the controller which can giving a command to go *up*, *down* or the *neutral* command. In figure 13 the behavior of a powerwindow plant with an infrared sensor is shown. The basic states are the same as in the modular Petri net powerwindow without the infrared sensor, but additional states where an obstacle blocking window movement up is detected have been added.

The modular Petri nets in both figures 13 and 12 include the necessary ports for communication with the controllers. Notice that we could have chosen to represent the physical behavior of the powerwindows differently if a more more precise behavioral representation would be required. For example, we could have a finer representation of the position a window in its physical frame or a finer representation of obstacle detection by adding more intermediate places to the Petri nets presented in figures 13 and 12. In fact, we have chosen as example the infrared obstacle sensor which outputs a binary signal (obstacle detected or no obstacle detected). In order to generate a modular Petri

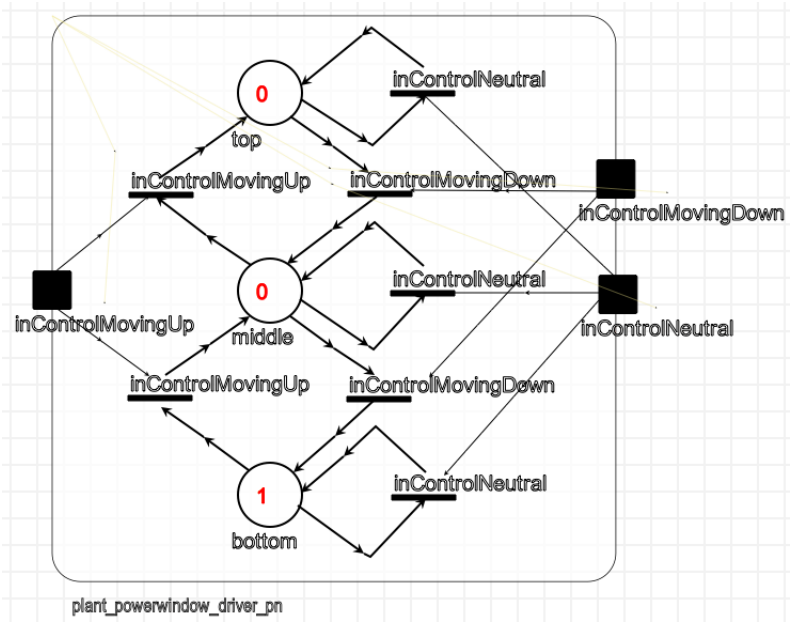

Figure 12: Transformed Driver Window Plant Model

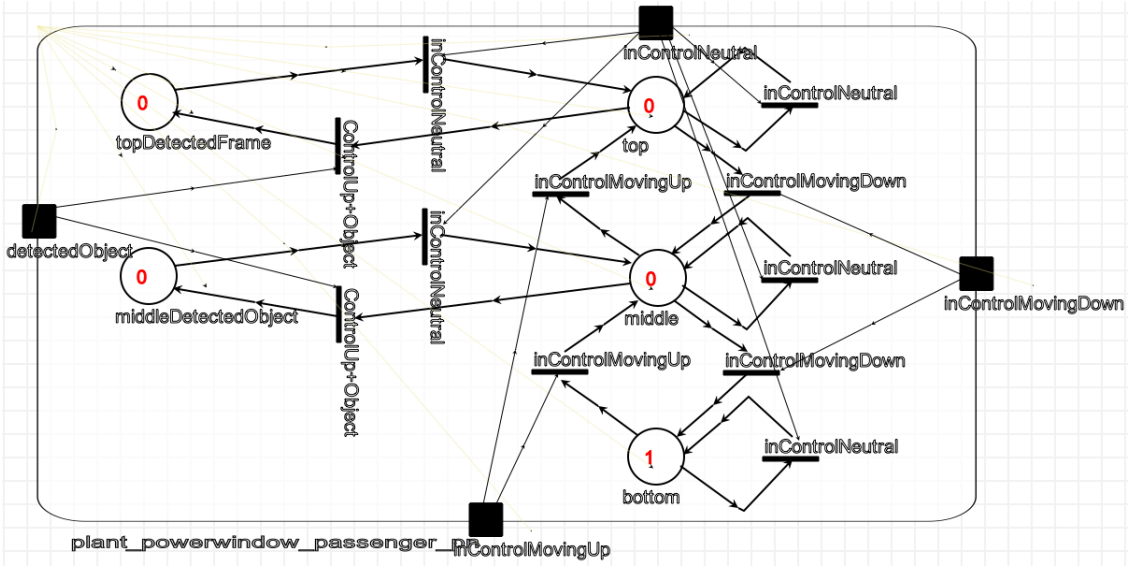

Figure 13: Transformed Passenger Window Plant Model

net representation of a powerwindow with a *force detecting* obstacle sensor we would need a representation of force detection which would be finer than the one required for the infrared sensor. This is so because power consumption values in the powerwindow electrical motor need to be monitored such that the window is not stopped unless sufficient resistance for is applied to the window.

3.2.3 Transformation of the Control Models into Petri Nets

Figures 15, 15 and 16 represent the modular Petri Net behavior of three controllers: the controller for a powerwindow switch without obstacle control, the controller for a powerwindow with obstacle control, and the controller for the lockout switch. Unsurprisingly, these are the switches that can used when building powerwindow plants.

Because the semantics of the simplified statecharts we have used to define the controllers can be easily simulated using Petri nets, the resulting modular Petri nets presented in figures 14, 15 and 16 are structurally very similar

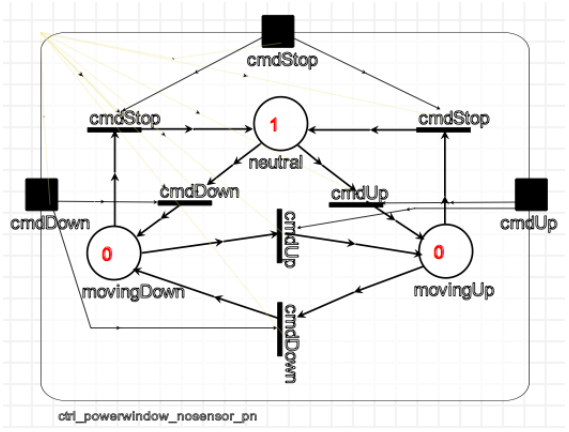

Figure 14: Transformed Window without Obstacle Detection Control Model

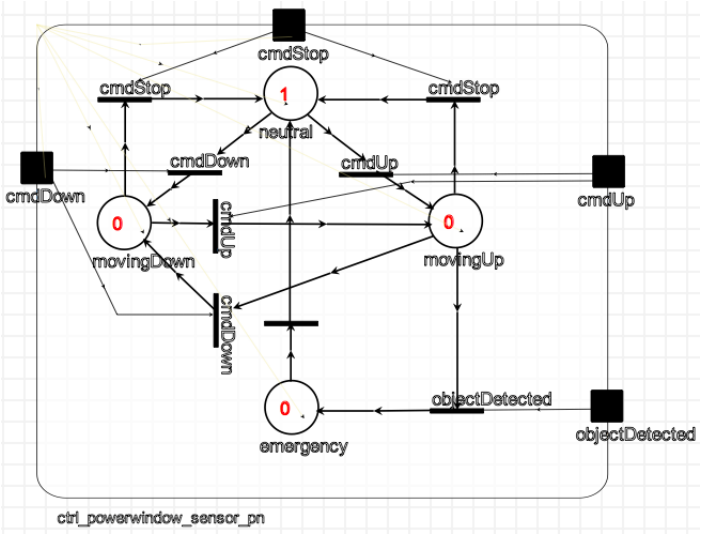

Figure 15: Transformed Window with Obstacle Detection Control Model

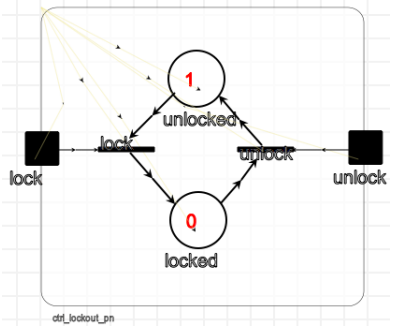

Figure 16: Transformed LockOut Control Model

to their counterparts in figures 7, 6 and 8 respectively – statechart *states* are transformed into modular Petri net *places* and statechart *transitions* are transformed into Petri net *transitions*. The modular Petri net's transitions are synchronized with module's *ports* having the same name.

3.2.4 Transformation of the Network Model

The network model we have presented in figure 9 connects the *environment*, *control* and *plant* domain specific components we have defined in section 3.1. The *network* at the domain specific level is transformed into a specific network that connects the modular Petri net components described in sections 3.2.1, 3.2.3 and 3.2.2. The result of this transformation is presented in figure 17.

At the domain specific level many details of the behavior of the modular components and how they are linked together is abstracted. This is so because the domain specific languages for the powerwindow are built to, in principle, allow the automotive engineer to be as expressive as possible using minimalistic domain specific constructs. For example, because the complete set of window movement ports is abstracted at the DSL level, they need to be expanded both at the component and at the *network* level. Also, in the domain specific network it is abstracted how the controllers send signals to each of the powerwindows defined in the plants. As can be observed in figure 17 this issue is tackled in our example by connecting the appropriate kind of controllers to the modular Petri net generated plants for the two powerwindows present in the plant in figure 2.

3.2.5 Composition Transformation

The composition transformation builds the connections between the modular Petri nets obtained in sections 3.2.1, 3.2.3 and 3.2.2, by using the network model in figure 17. For reference, we present in figure 18 the set of petri Nets that are to be composed, without their communication ports. In figure 19 we present the composed version of figure 17. Note that, not to overwhelm the reader, in figure 19 we provide only some of the composition links between the individual nets.

3.3 Simulation Activities

In this section, the generation of a hybrid simulation model from the DSL models in section 3.1 is described. The hybrid simulation is composed out of the Causal Block Diagram formalism and the statechart formalism. Causal Block Diagrams (CBD) are a general-purpose formalism used for modeling of causal, continuous-time systems. CBDs are commonly used in tools such as MathWorks Simulink®. CBDs use two basic entities: blocks and links. Blocks represent (signal) transfer functions, such as arithmetic operators, integrators or relational operators. Links are used to represent the time-varying signals transmitted between connected blocks. The physical components of the system and the environment are described using this continuous-time formalism. On the other hand, the controller is described in a discrete-event based formalism hence the name hybrid simulation model. Different approaches are possible for the composition and execution of hybrid simulation models: (a) the creation of super-formalism, (b) transformation to a common formalism and, (c) co-simulation.

The goal of generating a hybrid simulation model is to check certain functional properties of the interaction of the control software with the physical plant. For example that the window is lowered 10 cm on detection of an obstacle or to evaluate that the force on the object does not exceed a certain threshold. In this section we describe the generation of a hybrid model of the power window using a set of transformations. We however do not describe how this model is executed.

3.3.1 Transformation of the Environment Model into a Causal Block Diagram

In figure 20 we show the result of the transformation of the environment DSL model of figure 4 into a causal block diagram. The model is encapsulated in a *child* block that allows hierarchical modeling so it can be used as a whole when composing the full hybrid simulation model. The most important block involved in this model is a source block that generates a sequence of output values. This sequence is based on a vector containing tuples of time and output value. The block computes its output values based on a piecewise constant reconstruction of the signal based on the input samples.

The transformation creates a "sequence block" for each of the unique states in the environment model. The vector of tuples inside a created block is based on the time defined in each state of the environmental model. As a parameter, the transformation needs the translation between the event name (in the DSL) and the corresponding value that can be used within the Causal Block Diagram. It also needs a default value when the event is not applied.

The generated model is encapsulated in a child-block. The child block encapsulates a set of blocks, replacing them by a single block. It is used to hierarchically model the system and reduce the visual complexity. A child block can have multiple input and output ports. These ports are linked to the in- and output-blocks within the submodel. The model in figure 20 outputs all the signals to the parent model using the output ports.

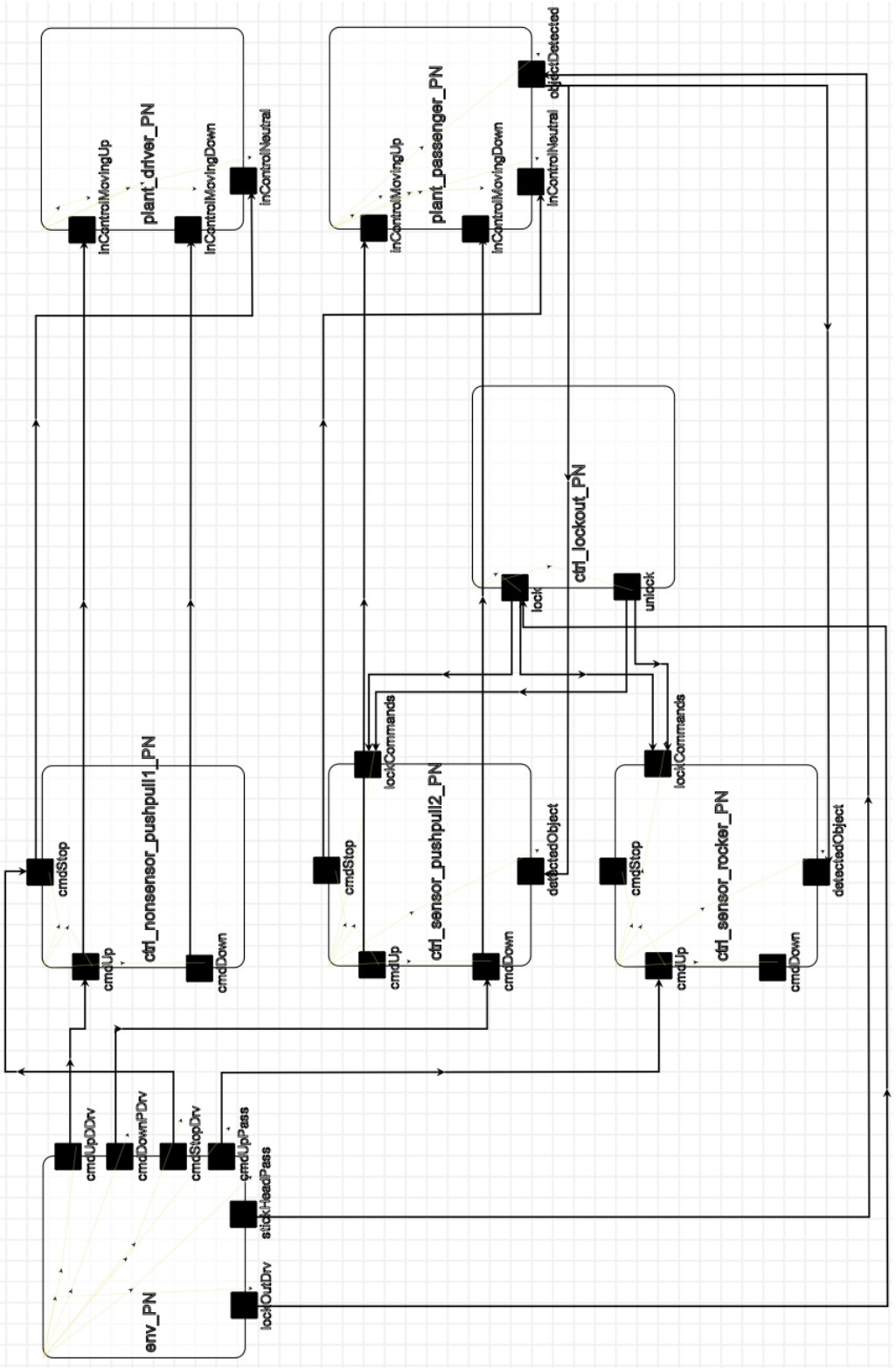

Figure 17: Transformed Network Model

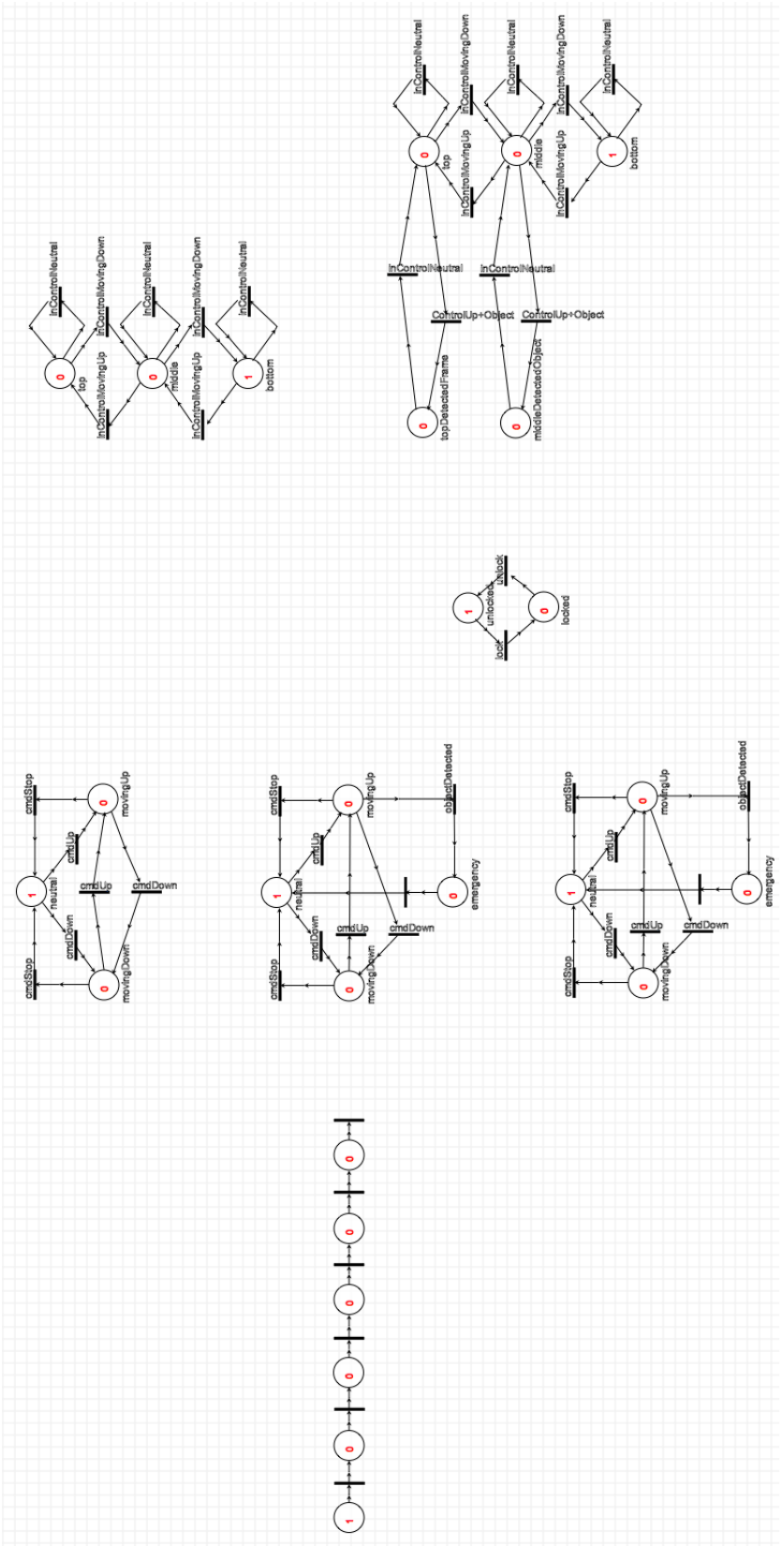

Figure 18: Uncomposed Petri Net Model of the Powerwindow Software

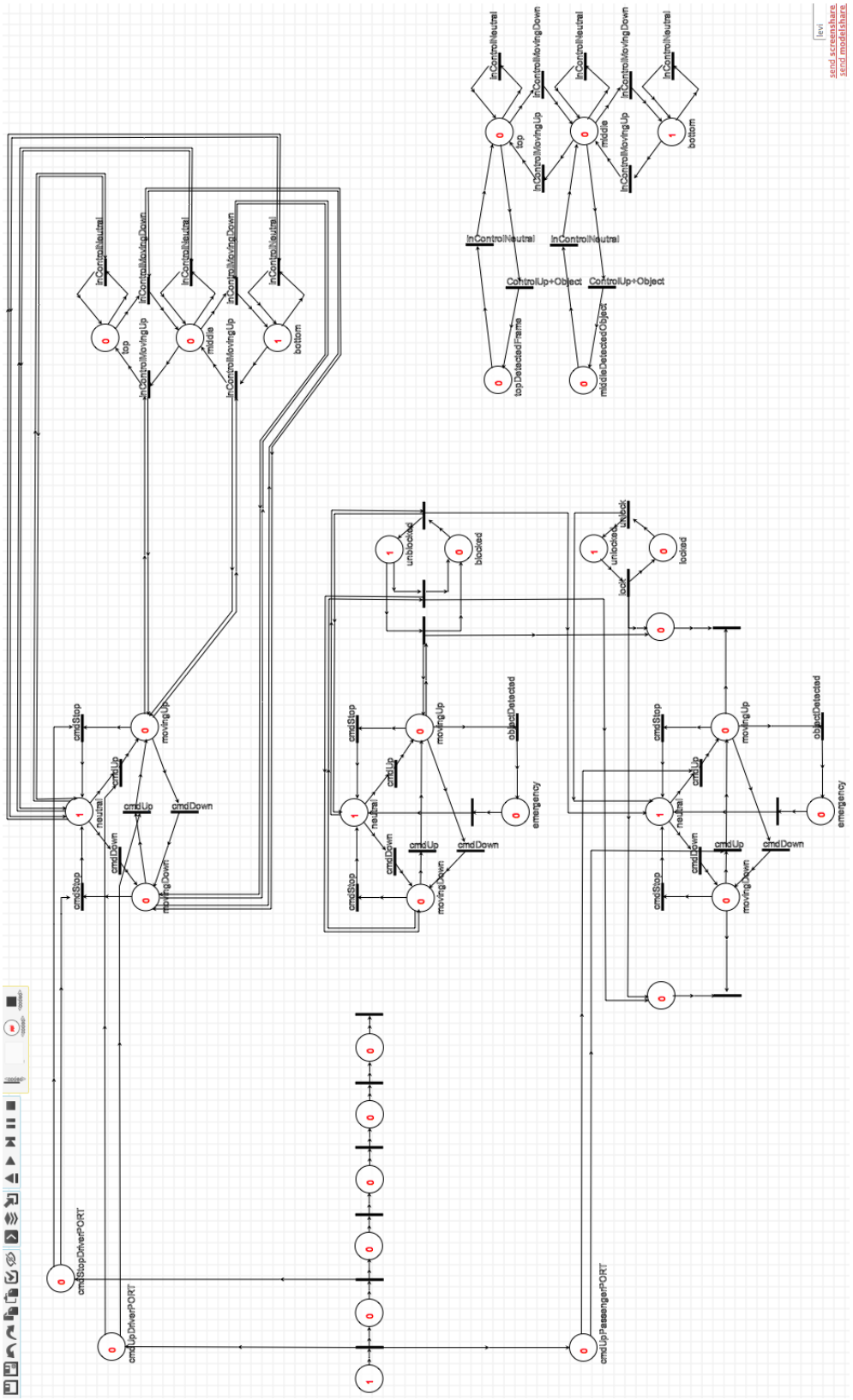

Figure 19: Transformed Composed Petri Net Model of the Powerwindow Software

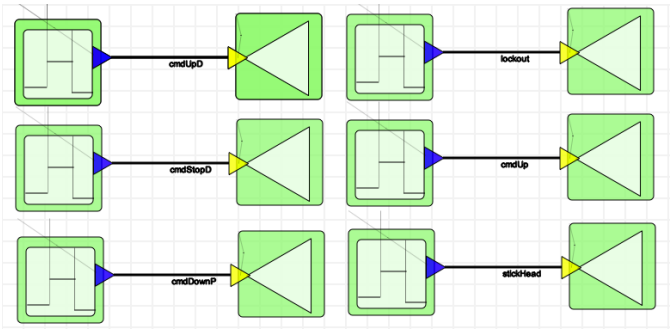

Figure 20: Causal Block Diagram of the environment model

3.3.2 Transformation of the Plant Model into a Causal Block Diagram

The power window can be transformed into a continuous model of the behaviour of the up and down movement of the window. Like in section 3.2.2 two models are generated from the DSL model. Both transformations use information like window height, motor gain and window friction from the DSL model.

Figure 21 represents the model of the window without any obstacle detection. The model is a simple second order model of the up- and downward-movement of the window. It outputs the position of the window so it can be observed during simulation. As with the environment model, it is encapsulated in a child block. If the window is not on top or bottom, the input and output commands for the up- and down-command are added together to get a single up or down command for the motor. This is multiplied with a motor gain while the friction is subtracted via a feedback loop. After integration of the acceleration we obtain the window speed from the input. The window speed is integrated to get the window position. bit better.

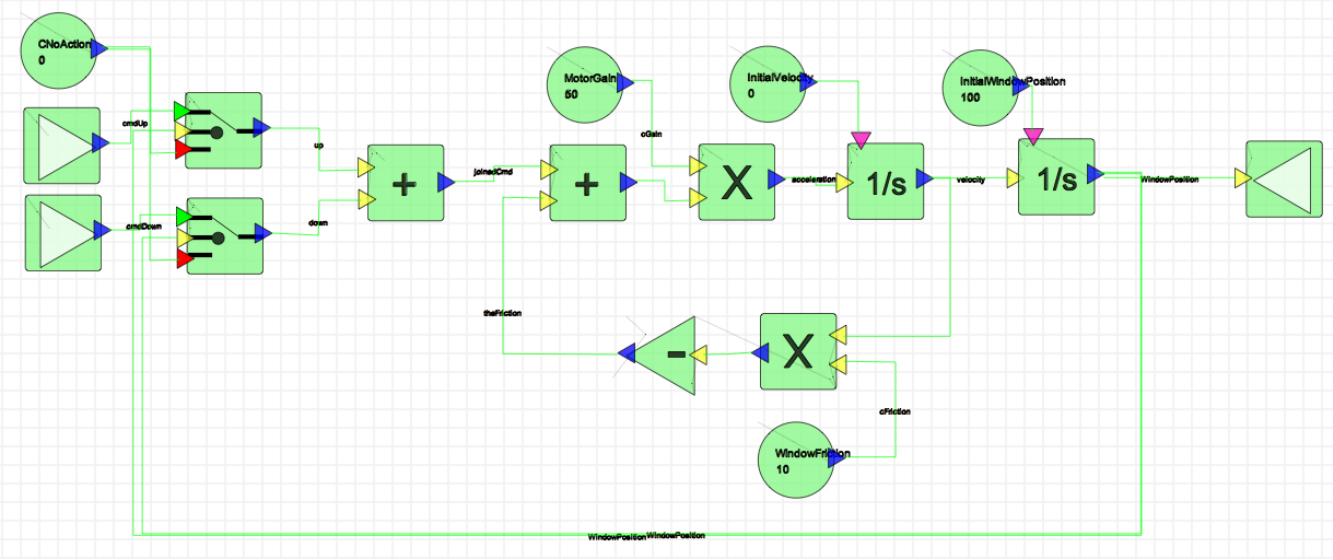

Figure 21: Model of power window plant without an obstacle sensor

The window with simple obstacle detection is shown in figure 22. The logic of the model is similar to Figure 21. The biggest difference is the output of the detected object when both an obstacle is present and the window is moving up.

More refined models can be used to assess more detailed behaviour of the window or when more complex sensors are used (for example to measure the force put on the window). These model can contain behaviour of gear and lever ratios, joints and other mechanical or electromechanical components.

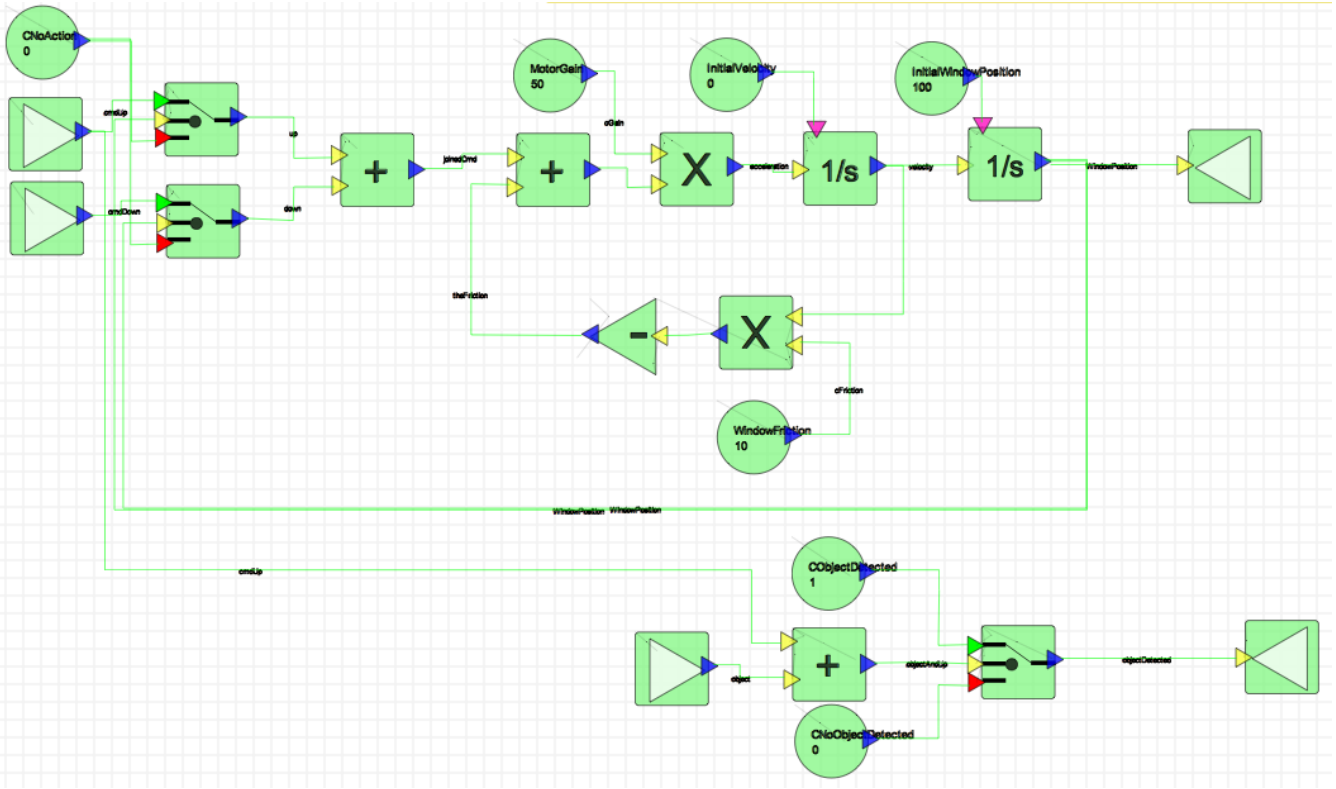

Figure 22: Model of power window plant with an object detection

3.3.3 Encapsulation of the Control Models

To allow the statechart to be used in a Causal Block Diagram, it needs to be encapsulated. The encapsulation of the statechart requires 3 blocks as seen in figure 23. The first is the State Event Locator (SEL). This block translates incoming signals into events so they can be fired by the statechart. The second block contains the statechart defined in 7. Finally actions have to be translated back to the continuous domain by the transducer block. Both the SEL and transducer block contain a lookup table that require input from the modeler.

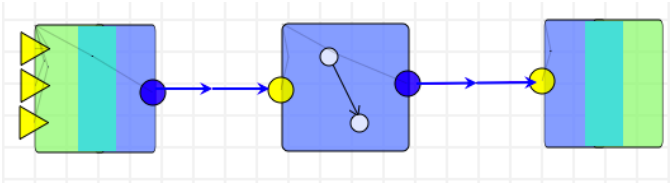

Figure 23: Encapsulated control

3.3.4 Transformation of the Network Model

The transformation of the network model is similar to the transformation defined in section 3.2.4.

3.3.5 Composition Transformation

The composition transformation composes the different models obtained in sections 3.3.1,3.3.2 and 3.3.3 using the network model. The name of the ports of the different child blocks match the name of the ports in the network model and can thus be matched easily.

When a component in the network diagram has multiple input components on the same port, a new component needs to be created. This component, the arbiter, decides the value to pass in case of a conflict. Since the value to pass can depend on all incoming ports of the component, the arbiter is set between this component and all its

incoming components. The content of this arbiter can be defined in CBDs or statecharts. Depending on the choice, this should also be encapsulated. In our case the motor commands of the passenger window have multiple inputs. The arbiter is set in between all the incoming control models of the passenger window side.

The composed model can be seen in figure 24. Since the formalism need input values on all of the defined input ports, the unconnected ports need to be connected to a default value. In this case the non issued commands for the controller parts are given a no action value.

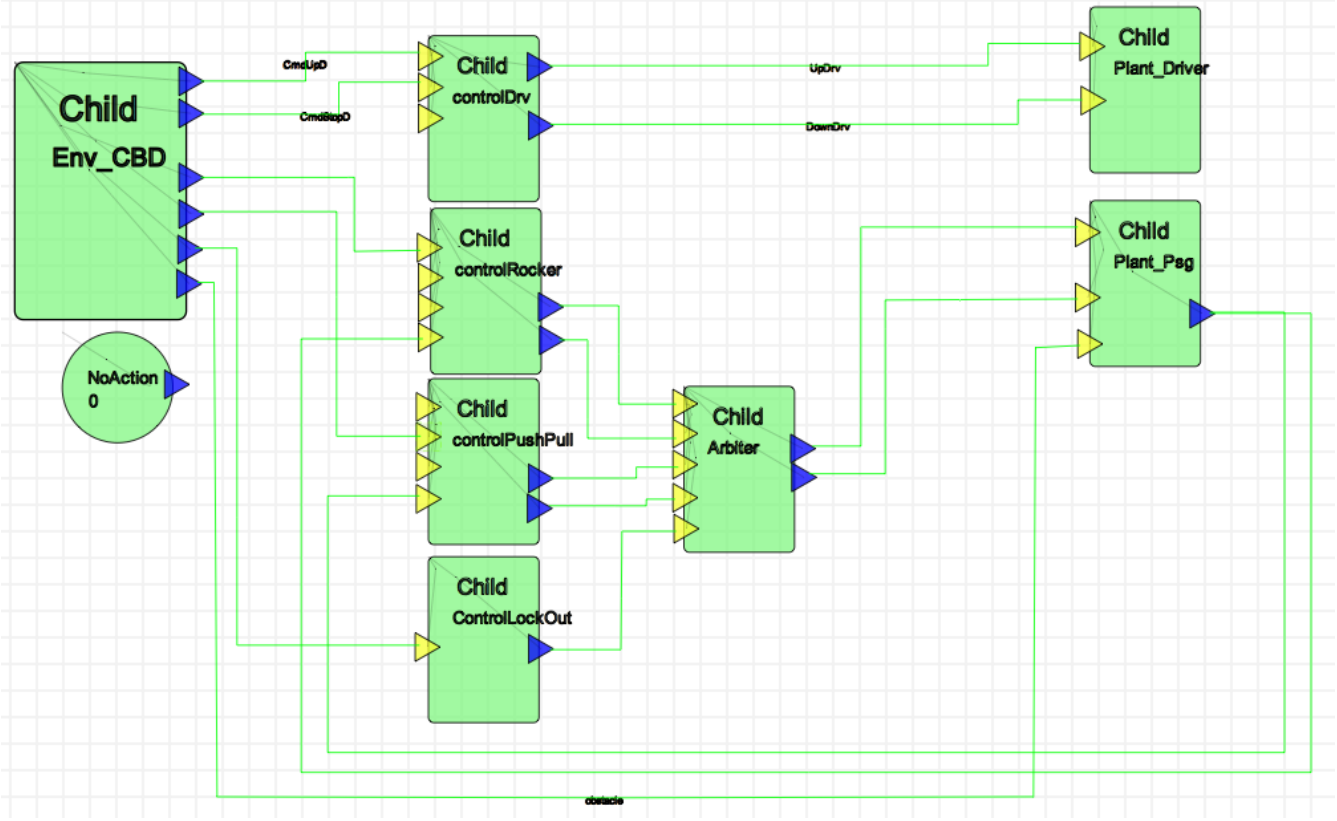

Figure 24: Composition of the full hybrid simulation model

3.4 Deployment Activities

This section is devoted to the exploration of the deployment space. During the process of deployment onto hardware a plethora of configuration choices have to be made in the middleware. These choices range from the mapping of software components to a hardware platform. But also lower level decisions like mapping of software functions onto tasks and assigning these tasks a priority. On the network side we have similar choices like the mapping of signals to messages and low level parameters that affect the sending and receiving of messages on the bus.

As a deployment platform we will use the AUTOSAR platform. To keep complexity under control and to create a competitive market for automotive software components, some leading automotive companies created the AUTOSAR consortium [3]. The AUTOSAR technical goals include modularity, scalability, transferability and reusability of functional components. To achieve these goals, the AUTOSAR initiative has a dual focus. On the one hand it defines an open platform (middleware) for automotive embedded software through standardized interfaces. On the other hand it provides a method to create automotive embedded systems. Using AUTOSAR, software can be developed mostly independently from the platform it will be deployed on.

The deployment space exploration consists out of 4 independent parts: (a) converting the statecharts to an AUTOSAR software component diagram, (b) generation of a calibration infrastructure, (c) the deployment space exploration and (d) code generation activities.

3.4.1 Converting statecharts to an AUTOSAR software component model

The functional model of AUTOSAR consists of a set of *atomic software components*. These components can interact with each other using *ports*. The service or data provided or required by a port are defined by its *interface*. This can be either a data-oriented communication mechanism (sender/receiver interface) or a service-oriented communication mechanism (client/server interface). The data-oriented interface can support 2 types of semantics. The first is "last-is-best", where only the last received value is stored. The other is a queued version where the data is stored in a queue until it is read. Each software component defines its behaviour by means of a set of *runnables*. A runnable is a function that can be executed in response to *events*, for example from a timer or due to the reception or transmission of a data element.

From the statechart a software component is generated containing the logic in a single runnable. Though for every incoming port to the SEL block and outgoing port in the transducer block, a sensor-actuator block is created. These sensor-actuator blocks can access the hardware of the platform (for example an Analog-Digital converter or a general purpose input-output pin). The logic in these blocks contains a function to: 1) translate the electrical signal values coming from the sensor to an engineering value; or 2) on the actuator side to convert the engineering value to an electrical signal.

Other information must be added as well this includes the events triggering the components and datatypes exchanged between the components have to be set.

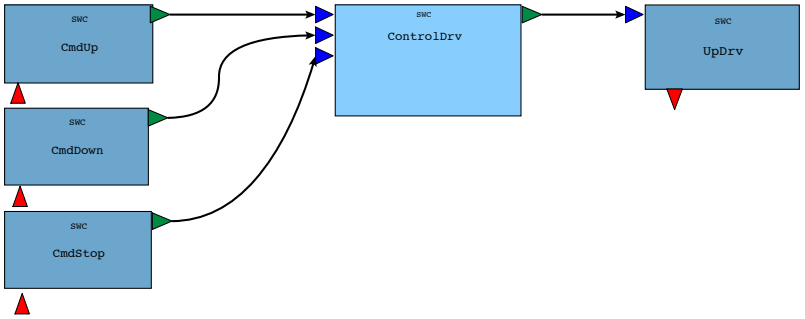

Figure 25: AUTOSAR software component model

figure 25 shows the generated AUTOSAR component diagram of the driver side. The passenger part is omitted from the model. All the software components contain a single runnable. Since the hybrid simulation model was executed in a 1 ms loop, the software functions are triggered every milisecond.

3.4.2 Calibration infrastructure generation

The exploration of the deployment space relies heavily on simulation and analysis. A crucial step in this process is the calibration of the models involved. Parameters to be estimated are for example network throughput, memory consumption and execution time of the different software components. This information can be obtained with data-sheets, low-level simulation or with the actual hardware. The state of art obtains the values using a combination of execution and analysis. Today, the state of the art obtains the calibration parameters by instrumenting application source code to record the execution times (or other parameters) of the different components and executing them on either the real hardware or on a cycle-true simulation of the hardware. Calibration can be done either before or during simulation. Since a software system is composed of multiple software components, the output value of a component can be propagated to a downstream component, so it does not need to be provided explicitly. Input components however still need an input that reflects the actual operation of the system under different operational conditions. However pure analytical methods are also available in the real-time community based on abstract processor models.

For calibration of performance models of cyber-physical systems, that have a tight combination of the physical and computational components of the system, the input values of the software input components originate in the environment of the system and in the feedback loops that exist between the computational and physical components. As a consequence, a trace-driven approach to supply the input components with input signals is not feasible due to the effects of the software on the physical components and vice versa. Using the models developed in the previous section, we can generate a calibration infrastructure to measure the execution times, memory consumption, energy consumption, etc.

In our example we will use the target hardware to run the computational components of the system while using a host computer to execute a the simulation models of the environment and plant. Signals generated by the environment and plant are transmitted by a bus, for example a serial connection, to the target board. The target board executes the computational components while measuring the calibration parameters. These are transmitted back to the host computer together with the output values that are used by the plant model. To synthesise this infrastructure three transformations are used. The models and code run in virtual time where the host computer is used to synchronize the time between the models. This is not a problem since we are interested in the properties of the basic blocks and not in the behaviour of the overall system.

Instrumented code can be synthesised from the AUTOSAR software components. These instrumentations measure certain properties of the AUTOSAR basic block. The instrumentation uses a call to a tiny middleware to read out sensors (like a timer, instruction counter, etc.). Also a small run-time environment is generated that will create the buffers for the communication signals and trigger the software functions at the right time. Also the basic blocks are executed in an atomic way so no effects from interrupts or preemptions can distort the measurement. In Listing 1 a small example of an instrumentation is shown.

Listing 1: Example instrumented code of a sensor-actuator component

```
{

    startMeasure1(); /*Instrumented code*/ status = CmdUp_RunRead();
    stopMeasure1(); /*Instrumented */ TxDistribution(__ID_CmdUpSensor_RunRead,
    getInterval1()); /*Instrumented*/
}
```

From the environment and plant models a simulator is generated. This is very similar to the generation of the hybrid simulation model in section 3.3.

Finally from the network model and hardware model the infrastructure is generated that captures, transmits and receives values on the host computer and target board. A template middleware is needed for each used target board involved in the hardware model.

When the measurements are collected from the calibration infrastructure, the results are annotated in a performance model. This performance model combines the type of processor with the software functions within a software component.

Execution Time (μs)	Distribution
20.000	7500
20.875	7499
21.375	1

Table 1: Example of a performance annotation between the ControlDrv and the MPC5567 hardware type

3.4.3 Deployment space exploration

In our example we will use an automatic deployment space exploration technique. It uses a platform-based approach by defining three different abstraction levels. At every level a transformation is defined to evaluate the real-time properties of the configuration.

At the first abstraction level the architecture is explored. The transformation maps the software components to the defined hardware components in the hardware model. It needs, as an input, a hardware model of the different components. Note that in more complex explorations this hardware model can also change. Changes include the number of hardware components, the type of processor, the type and number of communication buses, etc. figure 26 shows an example hardware model.

Since an AUTOSAR software component is atomic, it needs to be mapped to a single hardware component independent of the number of software functions in the software component. Sensor-actuator components are special since they need to be in the vicinity of their respective sensor or actuator. The system architect pre-maps these components. Normal software components can be mapped to any hardware platform. figure 27 shows a possible configuration of the mapping between the hardware model shown in figure 26 and the software component model in figure 25. Signals that are communicated between software components mapped to a different hardware component result in a signal transmitted on the bus.

The configuration can be evaluated using a bin packing check. As input it uses the performance model and architec-

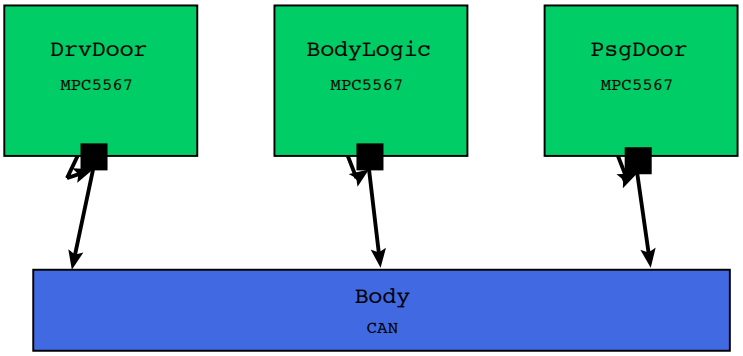

Figure 26: An example hardware model

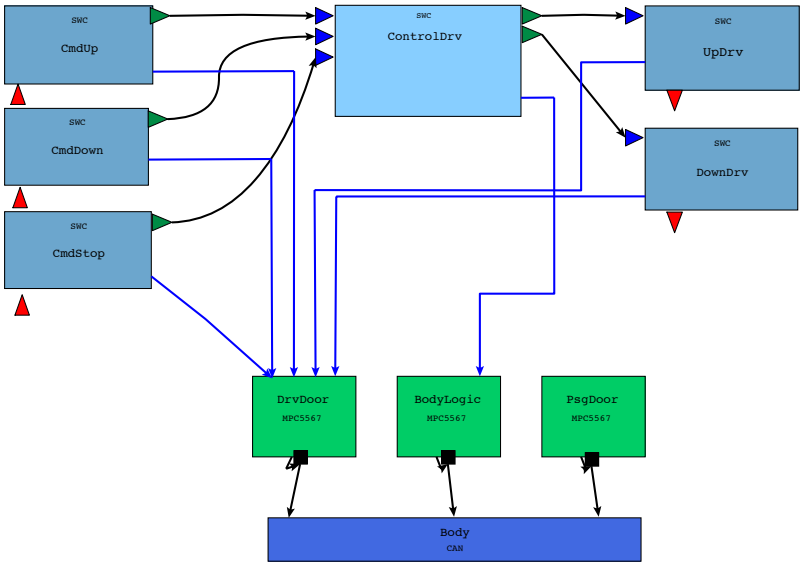

Figure 27: The software components mapped to the hardware model

ture model. The bin packing check is a simple algebraic equation to evaluate the usage of a hardware component. The algorithm calculates the smallest common multiple of the periods of the different software functions (functions with data-sent or data-receive events are assigned the period of their respective parent). For each hardware component the worst-case execution times of the mapped software functions are added (multiplied by the number of times executed in this time-frame). Dividing this time with the least common multiple is an indication of the usage of the hardware component. The same is done for the signals on the bus (without the overhead caused by frame headers and trailers). An example of the bin packing check for the BodyLogic component with the mapping of figure 27 can be seen below:

$$\ldots$$
$$\text{Usage of Processor BodyLogic} = ((1 * (21.375/1000))/1) * 100 = 2.1375\%$$
$$\ldots$$

The second step is the deployment exploration process is the mapping of the software functions to tasks on the operating system and assigning them a priority (the AUTOSAR operating system uses a fixed priority preemptive scheduler). Also depending on the bus type, the signals are mapped to messages on the bus and assigned a priority

(in case of an event-triggered bus) or a slot (in case of time-triggered bus). Properties of signals and messages are set. figure 3.4.3 shows a partial deployment model build using the eclipse modeling framework. The software functions are mapped to tasks. All properties of the task are set. The signals are also mapped to messages and assigned their properties.

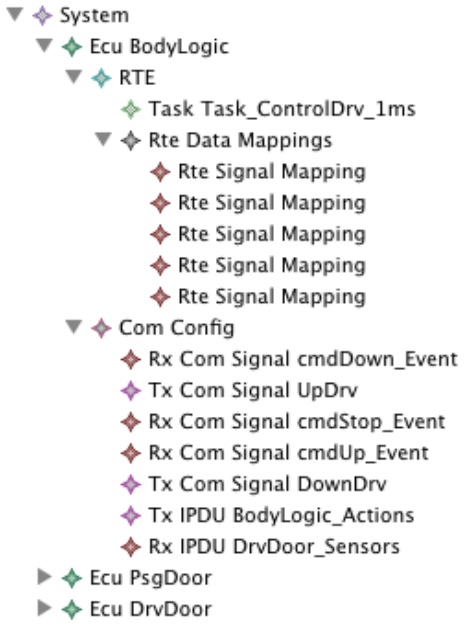

Figure 28: An example of a partial deployment. The example is made with an reduced version of the AUTOSAR meta-model in e-core.

Configurations at this level of abstraction can be checked using schedulabilty analysis.

Finally the low-level deployment starts. This is done by defining hardware buffers for the reception and transmission of messages. Since the hardware platform only has a limited amount of buffers in the communication controller the mapping is not a one-to-one mapping. The drivers and interfaces of the communication stack are configured and software buffers are defined if needed. Some hardware-specific options are also configured. figure 3.4.3 shows a full deployment configuration.

The last method of evaluation is a low-level deployment simulation. In our example we use a DEVS deployment simulation model. In Listing 2 a code snippet from an atomic DEVS model can be seen. The coupled DEVS of part of the deployment is shown in figure 3.4.3.

Listing 2: Example of an atomic DEVS model

```
class CanBusDEVS(AtomicDEVS):
    def __init__(self, name, speed):
        AtomicDEVS.__init__(self, name)
        self.state = CanBus(speed)
        self.INFRAMES = self.addInPort("CANFramesIn")
        self.NOTIFY = self.addOutPort("CANBusIdle")
        self.OUTFRAMES = self.addOutPort("CANOutFrames")
    def intTransition(self):
        self.state.onInternal()
        return self.state
    def extTransition(self):
        inFrame = self.peek(self.INFRAMES)
        self.state.onExternal(inFrame, self.elapsed)
        return self.state
    def timeAdvance(self):
        return self.state.getTimeLeft()
    def outputFnc(self):
        out = self.state.getOutput()
        if out is not None:
            self.poke(self.OUTFRAMES, out)
            self.poke(self.NOTIFY, out)
```

3.4.4 Code generation

The final step is the generation of the code that runs on the target platforms. This includes the generation of the middleware (adapted for the application), the application source code and the run-time environment to glue the

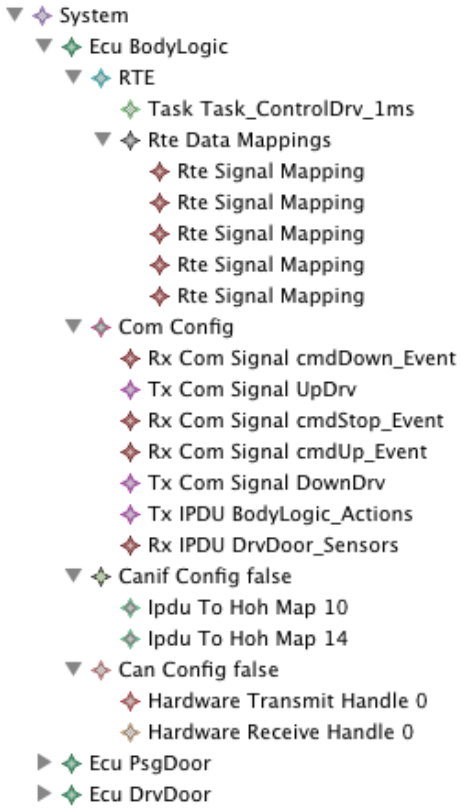

Figure 29: An example of full deployment. The example is made with an reduced version of the AUTOSAR meta-model in e-core.

middleware and application together. Details of this transformations can be found in the AUTOSAR specifications.

4 On the Transformations Required to Develop the Power Window

In this section we take a look at the transformations and their properties for building the power window system. The transformations are grouped in scenarios that focus on a single purpose. For each of these scenarios we first show a transformation graph. This graph shows visually the models (and their respective formalisms) and transformations between the models. Some transformations are shared among the different scenarios, in this case we will explain the transformation only the first time.

4.1 Scenario 1 – Safety Analysis

Figure 4.1 shows the models and transformations involved for creating the safety analysis model.

4.1.1 Environment DL DSL Models into Modular Petri Nets

Source: Environment DSL
Target: Modular Petri Net
Type Classifier: Exogenous, Vertical (refinement), Non-Parametric, Single Input, Single Output;
Rationale: The environment DL DSL allows declaring a set of activities being performed by multiple actors in a context for the *plant* and *control*. The transformation will give semantics to the environment DSL in Petri Nets by translating all the syntactically defined operators of the language as well as input and output events into communication with ports of the module;
Properties to Verify:

- Structural preservation in terms of the *Communication Sequence* blocks – sequences of communications will

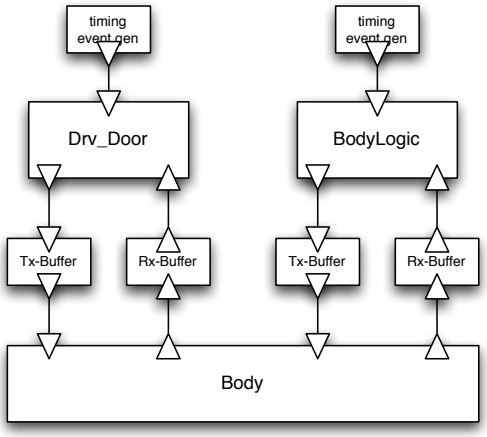

Figure 30: Example of a coupled DEVS model

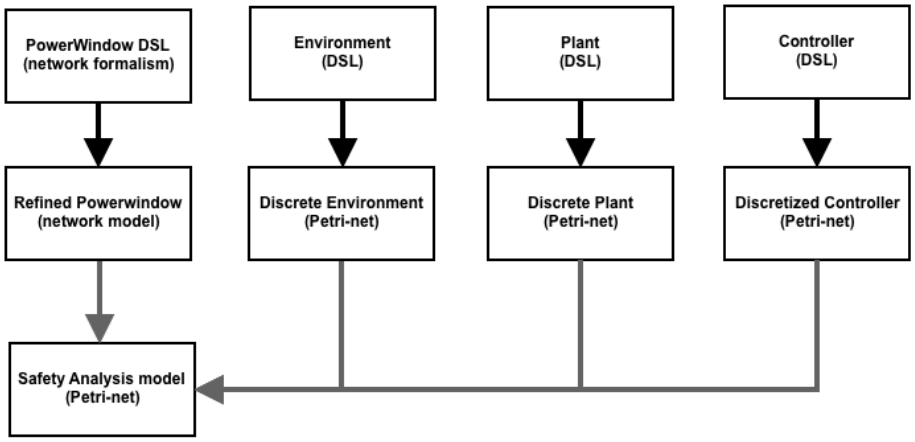

Figure 31: Transformation graph for generating the safety analysis model

correspond to sequences of transitions in modular Petri Nets;

- Correct generation of the semantics of the *parallel*, *sequential* and *alternative* operators;
- Correct generation of the module *ports* and correct connection of the generated Petri Net transitions with the generated *ports*.

4.1.2 Control DL DSL Models into Modular Petri Nets

Source: Control DSL
Target: Modular Petri Net
Type Classifier: Exogenous, Horizontal, Parametric, Single Input, Multiple Outputs
Rationale: The control DSL is a behavioural language, in the style of UML statecharts. Because of that the translation into Petri Nets is straightforward, with statechart states translated into Petri Nets places and statechart transitions translated into Petri Net transitions. Modularity information also has to be added. Takes as parameters the Plant model;
Properties to Verify:

- Structural preservation in the sense that all Control DSL states are transformed into modular Petri Net places, all transitions Control DSL transitions are transformed into modular Petri Net transitions, and their connections are rebuilt properly;

- Reachability preservation in the sense that all reachable states in the statechart should still be reachable in the corresponding places of the modular Petri Net;
- The resulting modular Petri Net is 1-safe;
- Correct generation of the module *ports* and correct connection of the generated Petri Net transitions with the generated *ports*.

4.1.3 Powerwindow (Plant) DL DSL Models into Modular Petri Nets

Source: Powerwindow DL
Target: Modular Petri Net
Type Classifier: Exogenous, Vertical (refinement), Non-Parametric, Single Output, Multiple Outputs
Rationale: The Powerwindow DL is a purely declarative language stating the hardware components of a power-window car setup. From this setup we generate plant models which define the discrete behavior in terms of window position of the powerwindows involved in the setup;
Properties to Verify:

- All the semantics are added by the transformation itself, so only the outputs need to be verified. In this case there is very little variability because a simple plant can be generated (without sensor), a plant with an infrared sensor or a plant with a force detecting sensor. The proof can be achieved exhaustively by showing that each of the three generated plants is the correct one;
- Correct generation of the module *ports* and correct connection of the generated Petri Net transitions with the generated *ports*.

4.1.4 Network DSL Models into Network PN Models

Source: Network DSL
Target: Network DSL
Type Classifier: Endogenous, Vertical (refinement), Parametric, Single Input, Single Output;
Rationale: The Network DL allows connecting components in the various DSLs. It does not have semantics other than the relations and their directionality. The semantics are given by the transformation itself parametrized by all the DSL models. The input network may be heavily modified, with new components at the control and plant level, with new ports and new relations being added. Takes as parameters all the powerwindow DSL models;
Properties to Verify:

- the newly created ports in the transformed network components should be correctly expanded from the initial window movement ports in the DSL model modules (in a macro expansion fashion); new ports in the control components and connections between those ports need to be created in case a lockout switch exists in the plant;
- the number of controller components with obstacle detection sensor to be generated should be the same as the number of powerwindows without obstacle detection sensor in the plant; likewise for the number of controller components without obstacle detection;
- the number of generated plant components should be the same as the number of powerwindows defined in the plant DSL model.

4.1.5 Composition of the obtained Petri Nets using the Network PN Model

Source: Modular Petri Nets, Network DSL
Target: Petri Nets
Type Classifier: Endogenous, Vertical (refinement), Non-Parametric, Multiple Inputs, Single Output;
Rationale: This composition transformation takes as input all the Modular Petri Nets models plus the network model generated from phase 1 and creates an integrated Petri Net with the whole system;
Properties to Verify:

- Structural preservation of the input Petri Nets;
- All structurally preserved components nets remain 1-safe;
- Reachability is preserved in each of the preserved component nets;
- Safety is preserved in the sense that no tokens are introduced in the token game of each component net;

- New parts of the net introduced by the transformation connect the components properly according to the network model. For example it is necessary to verify the dominant behavior of driver button controller of the passenger window towards the passenger button controller of the passenger window. Also, it is necessary to verify that the kind of composition performed is the one necessary one for each part of the net. For example, while the composition between the environment and the control components is achieved by introducing an intermediate place between connecting transitions, the composition between the control components and the plant is achieved by connecting control transitions to plant places.

4.2 Scenario 2: Hybrid Simulation

In Figure 4.2, the models and transformations for generating a hybrid simulation model are shown.

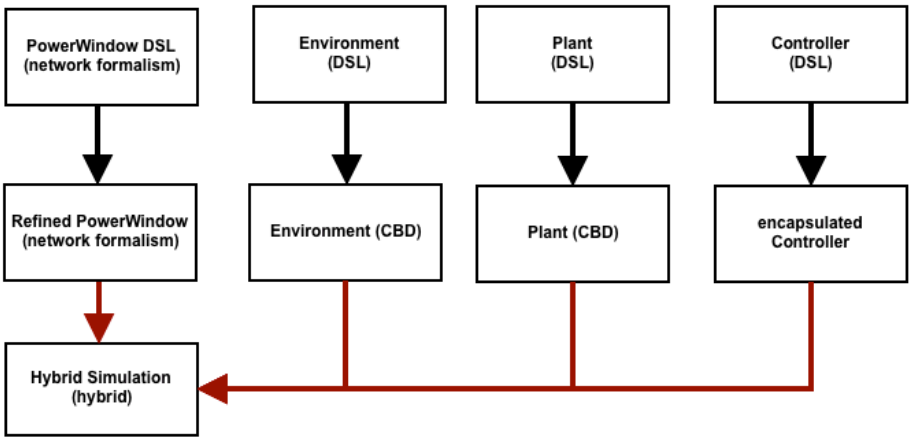

Figure 32: Transformation graph for generating the hybrid simulation model

4.2.1 Generate Continuous Power Window Environment Model from DSL model

Source: Environment DSL
Target: Causal Block Diagram
Type Classifier: Exogenous, Vertical (refinement), Parametric, Single Input, Single Output
Rationale: Continuous semantics of the environmental domain specific model. The different activities are translated to a signal value. Each activity requires a default signal value and an applied signal value;
Properties to Verify:

- Structural preservation in the sense that all unique activities are transformed into a sequence block;
- Correct generation of the time behaviour of the sequence block.

4.2.2 Generate Power Window Plant CBD Model from DSL model

Source: Plant DSL
Target: Causal Block Diagram
Type Classifier: Exogenous, Vertical (refinement)
Rationale: Denotational semantics of the power window plant domain specific model. The transformation uses window properties described in the DSL (motor gain, window height, friction, ...) to create a continuous model of the window behaviour;
Properties to Verify:

- The continuous semantics are added by the transformation itself, so only the outputs need to be verified. In this case there is very little variability because a simple plant can be generated (without sensor), a plant with an infrared sensor or a plant with a force detecting sensor.

4.2.3 Encapsulation of statecharts

Source: Control DL model
Target: encapsulated statechart
Type Classifier: Exogenous, Vertical (refinement)
Rationale: To use a statechart in a causal block diagram, continuous signals have to be translated to discrete events. In the opposite direction this is also true, actions need to be translated to (continuous) signal values. The state event location and transducer blocks are used for this purpose. They contain a lookup table to allow this translation;
Properties to Verify:

- Every statechart needs to be encapsulated by a child block including SEL and transducer blocks;
- Structural preservation between the Control DL statechart and the encapsulated statechart;
- There is a bijection between input events on the Control DL statechart and the SEL table inputs;
- There is a bijection between output events on the Control DL statechart and the transducer table outputs.

4.2.4 Network DSL Models into Network CBD Models

ditto section 4.1.4. Except, merged statecharts are shown as a single blocks and all ports should be connected.

4.2.5 Composition of the the models obtained from phase 1 using the Network CBD Model

Source: network DL
Target: hybrid CBD model
Type Classifier: Exogenous, Vertical (refinement)
Rationale: Composition of the full hybrid simulation model using the generated child blocks from phase 1. The connections are based on the refined network model;
Properties to Verify:

- Structural preservation of the number of child blocks;
- Verification that a import of a block has a single input;
- Verification that no unconnected imports exist;
- New parts of the model introduced by the transformation connect the components properly according to the network model. For example it is necessary to verify the dominant behaviour of driver button controller of the passenger window towards the passenger button controller of the passenger window. This is the verification of the arbiter component.

4.3 Scenario 3 – AUTOSAR software component model

Figure 4.3 shows the transformations and models involved for generating an AUTOSAR software component model.

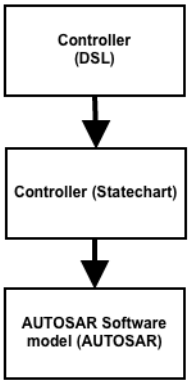

Figure 33: Transformation graph for generating the AUTOSAR component model

4.3.1 AUTOSAR Software Model

Source: Control models, network model
Target: AUTOSAR MM: Software part
Type Classifier: Exogenous, Vertical(refinement)
Rationale: The software parts of the developed models are transformed to the AUTOSAR MM so they can be used to be deployed on the AUTOSAR middleware;
Properties to Verify:

- Structural preservation: All logic components are translated to an AUTOSAR software component with a single runnable, including the arbiter components;
- Every incoming signal from the environment or plant and every outgoing signal to the plant models is translated to a sensor-actuator component.

4.4 Scenario 4 – Generate Calibration Infrastructure

In Figure 4.4 the transformation graph for generating the calibration infrastructure is depicted.

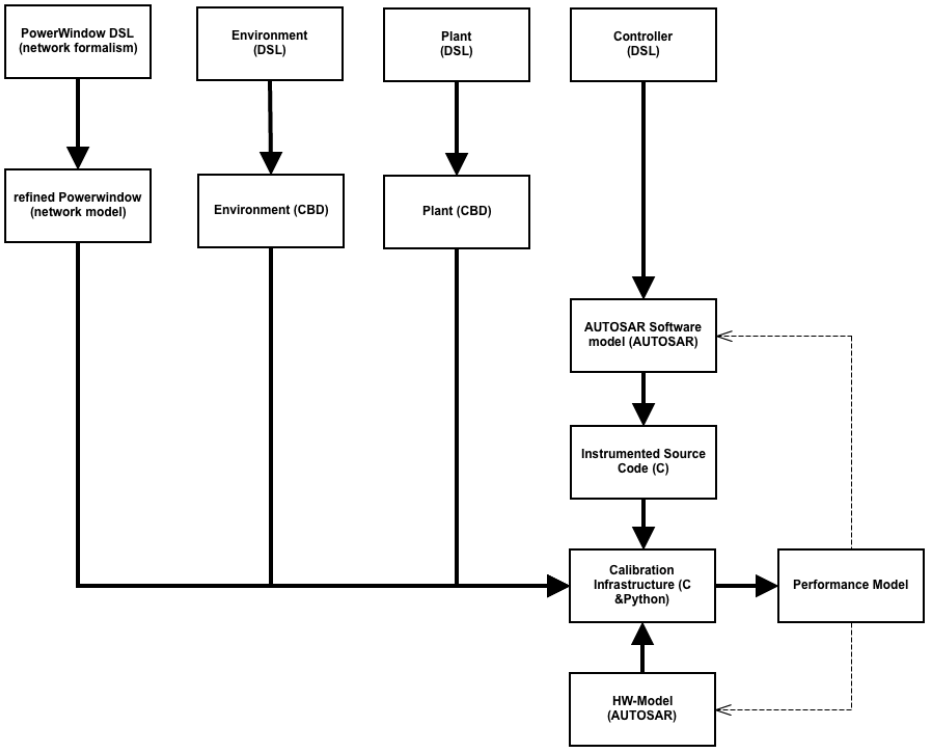

Figure 34: Transformation graph for generating a calibration infrastructure

4.4.1 Generate Instrumented Software Code

Source: AUTOSAR MM: Software part
Target: C-code
Type Classifier: Exogenous, Vertical(synthesis)
Rationale: C-code is generated for the software functions with instrumented calls, a small run-time environment;
Properties to Verify:

- Source and header file of the software function code (all software functions have a c-function) have the name of "softwareComponent_SoftwareFunction";
- An encapsulated call with instrumentation is generated (also in header file) with the name equal to "RTE_softwareComponent_SoftwareFunction";

- A function to call the instrumented software functions in the correct order (based on a precedence relation using a topological sort algorithm);
- For every receive port a buffer is created and rte functions to read and write this buffer.

4.4.2 Generate Plant Simulation Model

This transformation is equal to the transformation of Section 4.2.2.

4.4.3 Generate Environment Model

This transformation is equal to the transformation of Section 4.2.1.

4.4.4 Generate Infrastructure

Source: Network Model
Target: C-code, other code for interface depending on simulation tools
Type Classifier: Exogenous, Vertical(synthesis)
Rationale: The interface code is generated for the communication between host and target, and the time synchronisation;
Properties to Verify:

- Structural preservation of the network links between the involved components.

4.5 Scenario 5 – Deployment Space Exploration

The transformation graph of the deployment space exploration can be seen in Figure 4.5.

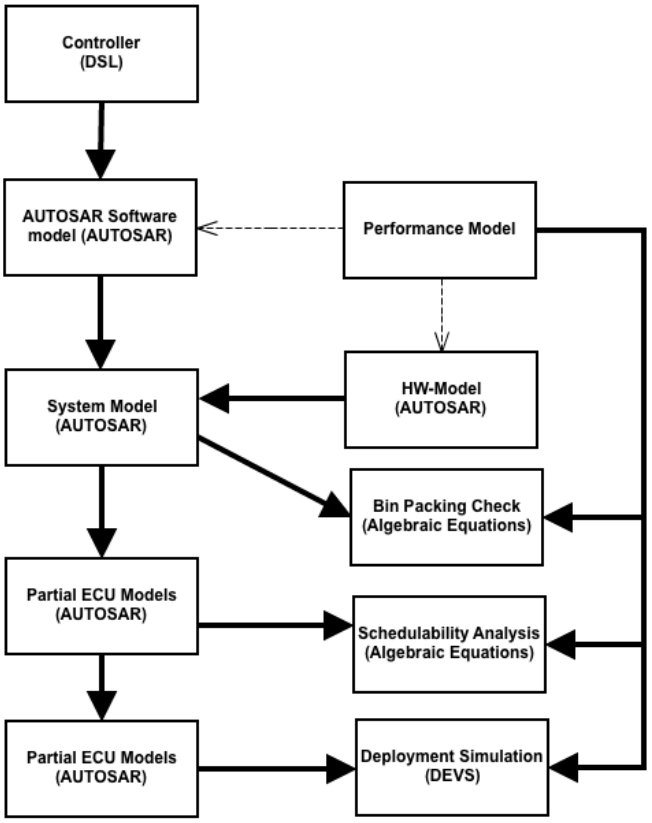

Figure 35: Transformation graph for exploring the deployment space

4.5.1 Map Software To Hardware (Architecture Space Exploration)

Source: AUTOSAR MM: Software part, AUTOSAR MM: Hardware part
Target: AUTOSAR MM: System part
Type Classifier: Endogenous, Vertical (refinement)
Rationale: This transformation maps software components to a hardware platform. Some components (especially sensor-actuator components) are already mapped because of the spacial requirements of the system, though unmapped components can be freely distributed over the available hardware platforms. The transformation can be used to explore the design space by using for example backtracking mechanisms in the transformation. To evaluate a solution a simple bin packing check can be used;
Properties to Verify:

- Structural preservation of both the software and hardware model;
- All software components are mapped to a single hardware component.

4.5.2 Bin Packing Analysis

Source: AUTOSAR MM: System part, Performance Model
Target: Algebraic Equations
Type Classifier: Exogenous, Horizontal
Rationale: The algebraic equation is a first check to evaluate the mapping of the different AUTOSAR application software components on the different hardware components. The equations adds the execution times of the different functions on a hardware platform within a certain time-frame. This time-frame is the smallest common multiple of the periods of all these different functions. The execution time of all the functions will exceed this smallest common multiple in a non-feasible solution;
Properties to Verify:

- There is a bijection between hardware components (ECU and Bus) and generated equations;
- Every software function in the software components is referred to once by one single bin packing equation.

4.5.3 Detailed Deployment Space Exploration (part 1)

Source: AUTOSAR MM: System Part
Target: AUTOSAR MM: ECU Part
Type Classifier: Endogenous, Vertical (refinement)
Rationale: In this transformation task and bus information is added. As with the architecture space exploration, this transformation can be used to explore the design space. The functions mapped to a specific controller are combined to tasks on the real-time operating system. The task is assigned a priority in the case of a priority scheduler or an execution slot when the RTOS is time-triggered. A similar procedure is done for the communication signals. These are combined into frames that can be transmitted on the bus. The frame is assigned a priority in case of an event-triggered bus like CAN or a transmission slot in case of time-triggered bus;
Properties to Verify:

- Every software function is mapped to a task with a (not unique) priority, functions mapped to the same ECU can be mapped to an existing task;
- Every signal is mapped to a message (with a unique priority or unique slot), signals originating on the same ECU can be mapped to an existing message;
- No message can exceed the maximum message size (defined by the protocol or parameters of the network);
- All properties of signals and messages are set (transmission modes, etc.).

4.5.4 Schedulability Analysis

Source: AUTOSAR MM: ECU part (Partial), Performance model
Target: Algebraic Equations
Type Classifier: Exogenous, Horizontal
Rationale: Schedulability analysis is a technique to assess the real-time characteristics of a software system. The transformation builds the equations for every task and message in the model;
Properties to Verify:

- Equation for every task and message defined in the model.

4.5.5 Detailed Deployment Space Exploration (part 2)

Source: AUTOSAR MM: System part
Target: AUTOSAR MM: ECU part
Type Classifier: Endogenous, Vertical (refinement)
Rationale: When the system is schedulable other deployment decisions can be made like the number of hardware and software buffers to assign to a frame. Also low-level driver information is added. These decisions can be evaluated using a full deployment simulation;
Properties to Verify:

- Every message transmitted or received on the ECU has a mapping to a buffer;
- All properties of the low-level drivers and interfaces are set.

4.5.6 Full Deployment Simulation

Source: AUTOSAR MM: ECU Models (Full), Performance model
Target: DEVS Simulation Model
Type Classifier: Exogenous, Horizontal
Rationale: The full deployment solution can be evaluated using a simulation. As an example simulation model we use a DEVS-based AUTOSAR deployment performance model;
Properties to Verify:

- A DEVS processor model is created for each ECU, all parameters are set correct;
- A DEVS bus model is created for each bus, all parameters are set correct;
- A DEVS output HW buffer is created for each ECU that is sending messages. The size and behaviour of the buffer depends on the parameters of the model;
- A DEVS input HW buffer is created for each ECU that is receiving messages; The size and behaviour of the buffer depends on the parameters of the model;
- All ports are connected.

4.6 Scenario 6 – Code Synthesis

Figure 4.6 shows the needed transformations to generate the code of the different ECUs.

4.6.1 Generate Application Software Code

Source: AUTOSAR MM: Software Component Part
Target: C-code
Type Classifier: Exogenous, Vertical(synthesis)
Rationale: generate the application source code for the ECUs;
Properties to Verify:

- For each defined function a c-code function is created;
- Naming conventions of AUTOSAR RTE are properly followed.

4.6.2 Generate Run-time Environment Code

Source: AUTOSAR MM: system part
Target: C-code
Type Classifier: Exogenous, Vertical(synthesis)
Rationale: Generation of the glue between the middleware and the application. This contains the code to trigger the execution of the runnable and the buffers of the signals. A specific RTE is generated for every ECU;
Properties to Verify:

- For each RTE code is generated;
- Buffers and access function are created only for the receive signals following the RTE naming conventions;
- RTE code is created for every transmitted signal (internal and external) following the RTE naming convention;
- RTE trigger code is created for every function mapped to the ECU following the naming convention.

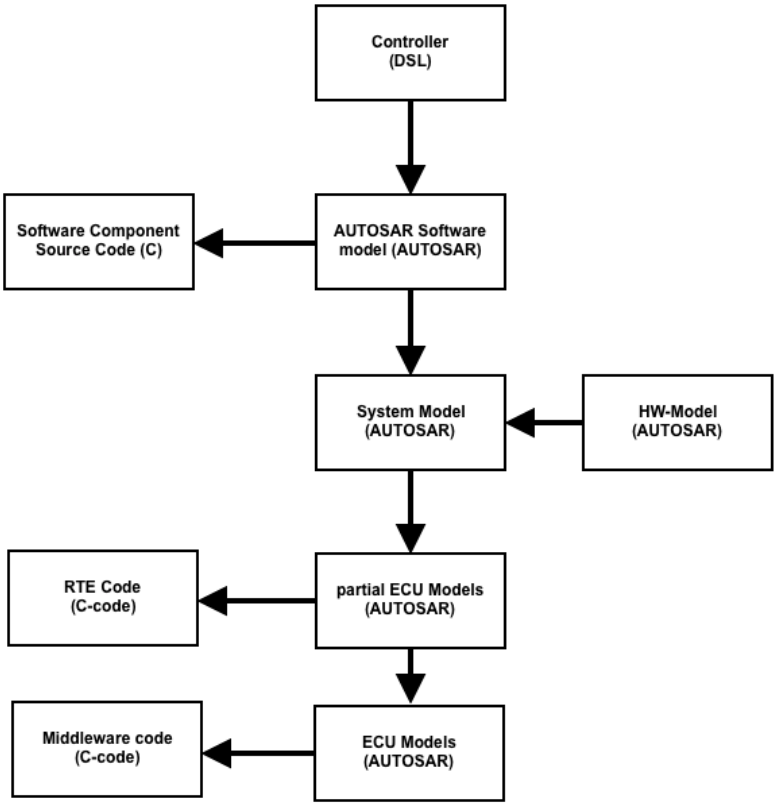

Figure 36: Transformation graph for synthesis of the ECU code

4.6.3 Generate middleware code

Source: AUTOSAR MM: ECU part
Target: C-code
Type Classifier: Exogenous, Vertical(synthesis)
Rationale: This generates a tailored middleware for each ECU.
Properties to Verify:

- For each ECU defined, the middleware code is generated;
- For every module referenced in the ECU module, code is generated.

4.7 Putting it all together

In Figure 4.6.1 all the transformations discussed can be seen.

References

[1] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. pages 243–258. Springer, 2002.

[2] ATLAS. ATLAS transformation language, 2008. `http://www.eclipse.org/m2m/atl/`.

[3] AUTOSAR. Official webpage. `http://www.autosar.org`, 2010.

[4] S. Cook, G. Jones, S. Kent, and A. C. Wils. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

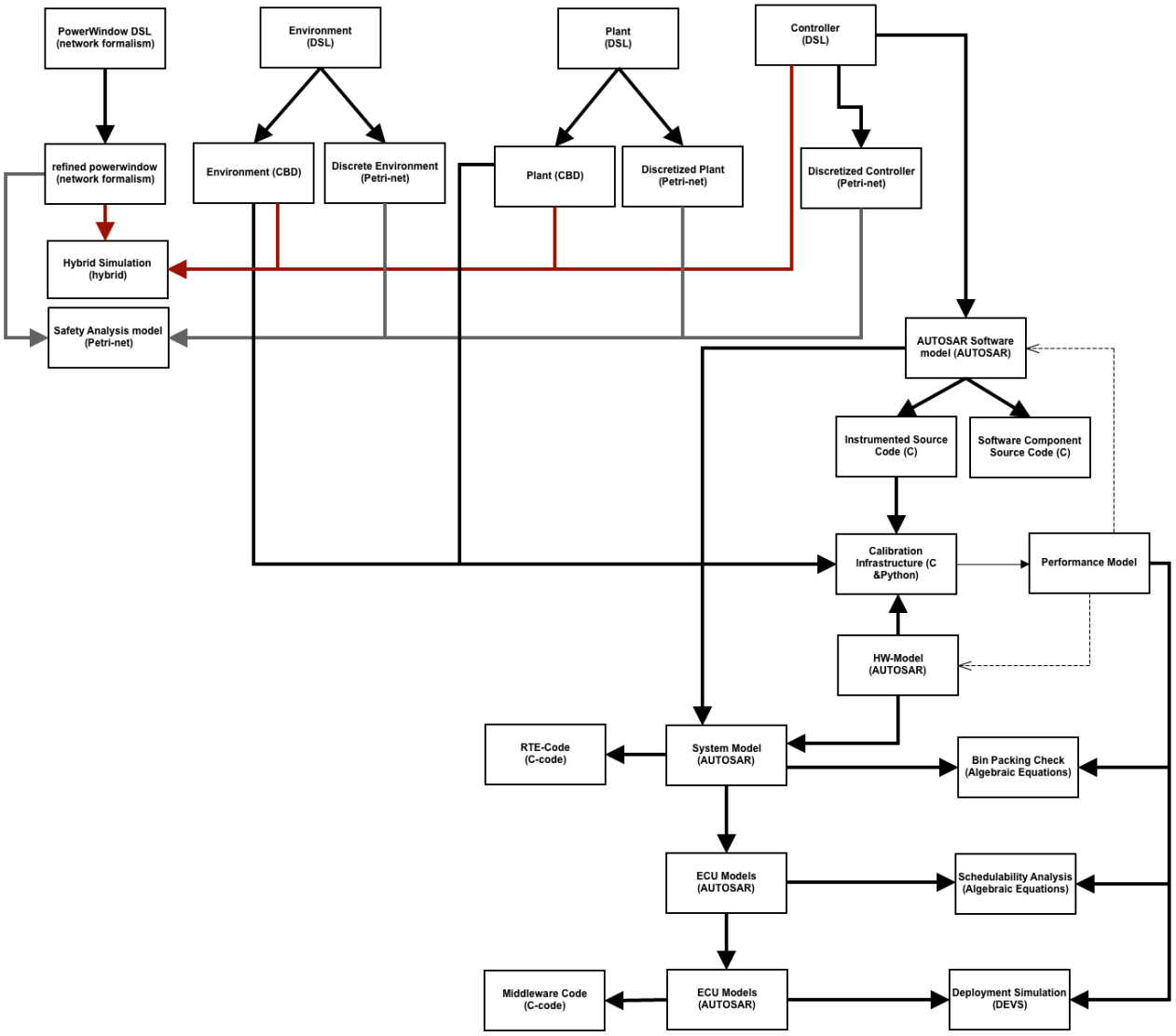

Figure 37: The full transformation graph

[5] J. de Lara and H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer-Verlag, 2002.

[6] P. Dhaussy and J.-C. Roger. Spécification du langage CDL v.1 : Syntaxe et sémantique (documentation provisoire). Technical report, LISyC, ENSTA Bretagne, 2011.

[7] R. C. Dorf. *Modern Control Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 12th edition, 2011.

[8] G. Dupe, M. Belaunde, R. Perruchon, H. Besnard, F. Guillard, and V. Oliveres. SmartQVT. `http://smartqvt.elibel.tm.fr/`.

[9] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] L. Lúcio, B. Barroca, and V. Amaral. A technique for automatic validation of model transformations. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 136–150. Springer-Verlag, 2010.

[11] Metacase. Domain-Specific Modeling with MetaEdit+: 10 times faster than UML. 2009.

[12] W. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM RedBooks, February 2004.

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

[14] A. Narayanan and G. Karsai. Towards verifying model transformations. *Electron. Notes Theor. Comput. Sci.*, 211:191–200, April 2008.

[15] A. Narayanan, G. Karsai, C. Ermel, R. Heckel, J. de Lara, T. Margaria, J. Padberg, and G. Taentzer. Verifying model transformations by structural correspondence. *Electronic Communications of the EASST*, 10, 2008.

[16] A. Raji, P. Dhaussy, and B. Aizier. Automating context description for software formal verification. In *MoDeVVa Workshop*, Oct. 2010.

[17] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20:42–45, September 2003.

[18] Transport Canada. Technical standards document no. 118, revision 1, power-operated window, partition, and roof panel systems, 2009. `http://www.tc.gc.ca/eng/roadsafety/safevehicles-mvstm_tsd-tsd-1180rev1_e-758.htm`.

[19] US Department of Transportation. RIN 2127-AG36 federal motor vehicle safety standards; power-operated window, partition, and roof panel systems, 2004. `http://www.nhtsa.gov/cars/rules/rulings/safety_switch/SaferSwitchesFinalRule.html`.

[20] D. Varró and A. Pataricza. Automated formal verification of model tranformations, 2003.