Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

THE MODELVERSE: A TOOL FOR MULTI-PARADIGM MODELLING AND SIMULATION

Yentl Van Tendeloo

Department of Mathematics and Computer Science
University of Antwerp
Middelheimlaan 1
Antwerp, BELGIUM

Hans Vangheluwe

Department of Mathematics and Computer Science
University of Antwerp / Flanders Make vzw
Middelheimlaan 1
Antwerp, BELGIUM

ABSTRACT

Multi-Paradigm Modelling (MPM) has been proposed to tackle the complexities found in Cyber-Physical Systems. MPM advocates the *explicit* modelling of all pertinent parts and aspects of complex systems. It adresses and integrates three orthogonal dimensions: multi-abstraction modelling, concerned with the (refinement, generalization, ...) relationships between models; multi-formalism modelling, concerned with the (multi-view, multi-component, ...) coupling of and transformation between models described in different formalisms; explicitly modelling the often complex, concurrent workflows. Current modelling, analysis and simulation tools support only isolated parts of MPM. The core methods and techniques enabling MPM are modelling language engineering, model operations (such as transformation and simulation), and workflow modelling. This paper delves into each enabler, presenting its relation to MPM and how it is supported in our prototype tool: the Modelverse. An automotive power window example is used to illustrate the Modelverse's capabilities. All aspects are explicitly modelled and enacted with a Formalism Transformation Graph + Process Model (FTG+PM).

1 INTRODUCTION

Complex Cyber-Physical Systems (CPS) consist of a physical part which interacts with its environment, are controlled by (embedded) software, and are often networked with other Cyber-Physical Systems. Modelling is essential to develop these systems and tackle their inherent complexity. In particular, Multi-Paradigm Modelling (MPM) (Vangheluwe, de Lara, and Mosterman 2002) proposes to explicitly model all relevant aspects of the system, using the most appropriate formalism(s), at the most appropriate level(s) of abstraction, while explicitly modelling the development process. Due to its nature, MPM spans a large number of domain-specific formalisms (as these are often most appropriate), with associated operations, combined in a process model. These aspects may be modelled using the Formalism Transformation Graph and Process Model (FTG+PM) formalism (Lucio et al. 2013).

Support for MPM, and in particular for the FTG+PM, is only partial in current tools. This, as MPM combines three research areas:

1. **Language Engineering** to create and instantiate new languages. These languages can be tailored to the problem domain, resulting in Domain-Specific Modelling Languages (DSMLs). This aspect of MPM lowers the cognitive gap between the problem and solution domain by decreasing verbosity and maximally constraining the modeller to the problem at hand.
2. **Model Operations** to define the semantics of models, and to execute them. These operations can be tailored to specific DSMLs, thereby giving semantics to user-defined languages. This aspect of MPM takes models beyond mere documentation, thereby increasing their usefulness.
3. **Process Modelling** to define the control and data flow of the development process. The process is tailored to a specific problem, which gives rise to causal dependencies between the used formalisms

and operations to map between them. This aspect of MPM provides an overview of the control and data flow used within the process, which can be used for documentation, analysis, optimization, and enactment (i.e., automatic chaining and execution of operations).

These three aspects are all present in the FTG+PM: language engineering specifies the nodes in the FTG, model operations specify possible mappings between these languages, and the process model is represented by the PM.

This paper explores the three aspects of MPM, and presents a prototype tool implementation,the Modelverse (Van Tendeloo 2015). The combination of language engineering, model operations, and process modelling with support for enactment in a single tool, makes our tool unique. Other tools only support at most two of these aspects, thus making it impossible to use a complete FTG+PM. We summarize each individual aspect, present why it is indispensible for MPM, and provide information on how our tool provides (additional) support. Each aspect is exemplified by means of our power window example, used throughout the paper. Additionally, our prototype tool runs as a service, and can therefore be accessed remotely by multiple users. The impact of this feature is also investigated.

For reasons of clarity, example models are shown visually, even though our tool does not (yet) support the visual representation of models. Our earlier prototype tools AToM^3 (de Lara and Vangheluwe 2002) and AToMPM (Syriani, Vangheluwe, Mannadiar, Hansen, Van Mierlo, and Ergin 2013) demonstrated the feasibility of visual modelling for MPM. Our tool and all relevant models can be found online at https://msdl.uantwerpen.be/git/yentl/modelverse.

The remainder of this paper is organized as follows. Section 2 presents necessary background. The following sections elaborate on the three aspects: language engineering (Section 3), model operations (Section 4), and process modelling (Section 5). Section 6 presents related work for each aspect, as well as how their combinations are supported. Section 7 concludes the paper and presents future work.

2 BACKGROUND

We first provide some of the necessary background for the remainder of this paper: Multi-Paradigm Modelling (MPM), the FTG+PM, and the example we are tackling.

2.1 Multi-Paradigm Modelling

Modern engineered, often cyber-physical, systems have reached a complexity that requires systematic design methodologies and model-based aproaches to ensure correct and competitive realization (Vangheluwe, de Lara, and Mosterman 2002). It is in this context that Multi-Paradigm Modelling (MPM) comes into play: multiple heterogeneous domains come together and need to be managed. MPM proposes to tackle these problems by modelling all relevant aspects of the system explicitly at the most appropriate level(s) of abstraction, using the most appropriate formalism(s), while explicitly modelling the process.

Models can then be easily used by domain experts, as they shield the accidental complexity of the underlying solution (*e.g.*, Petri net reachability analysis). Afterwards, models from different domains are (automatically) mapped to a common domain, which is appropriate for the problem at hand (*e.g.*, Petri nets). Individual models are interleaved, often automatically, without the need for any input from domain experts: their task is limited to modelling in a domain-specific language they are an expert in. This not only shields the domain experts from the underlying solution domain, but also makes the complete process repeatable, as it is explicitly modelled and merely enacted.

Multiparadigm techniques have been successfully applied in several domains (Vangheluwe, de Lara, and Mosterman 2002), proving that it is capable of reducing accidental complexity.

2.2 FTG+PM

With the large number of formalisms, models, and operations created during the application of MPM, it is easy to get lost. The FTG+PM (Lucio et al. 2013) was conceived to provide an overview of the complete approach, thereby providing an overview of all formalisms and how they are related (*i.e.*, the operations between them). Additionally, the FTG+PM keeps track of the different modelling artefacts created and the process itself.

As the name suggests, an FTG+PM model consists of two parts: the Formalism Transformation Graph (FTG) and the Process Model (PM). The FTG presents all used formalisms (the nodes) and their relations (the links), and presents an overview of all languages and operations used in the process. The PM mandates the order of operations to execute, as well as on which models (data) they operate.

An FTG+PM model has two primary applications: documentation and enactment. As documentation, the FTG+PM summarizes the different languages used, the required operations between them, and the process of getting from start to finish. Thanks to its simple visual representation, it can compactly summarize an approach. Enacting an FTG+PM means to execute the series of operations defined in the process model: start at the initial node and execute the operations in the defined order. Execution creates or modifies the models linked to that operation.

2.3 Running Example: Power Window

The power window example consists of a simple electronically controlled window of a car. Users control a button (the *environment*), which can be in three modes: up, down, or neutral. This button is connected to a controller (the *control*), which translates the keypresses into commands to the engine responsible for window movement. The window itself is raised by a wormgear (the *plant*), which can also be in three modes: up, down, or neutral. While there is an intuitive mapping between the controls and the window behaviour (*e.g.*, when the up button is pressed, the window should go up), there are some corner cases or additional requirements. Different requirements are used for different purposes in the literature (Denil 2013, Mosterman and Vangheluwe 2004). In this paper, we will focus on the verification aspect of the power window: we want to make sure that when an object is inserted through the window, the window will never exert a high power (*i.e.*, keep going up).

In the context of MPM, a power window is often used, as it is relatively minimal and easy to understand, while still posing many of the challenges faced in more general MPM problems (Denil 2013, Mosterman and Vangheluwe 2004).

An FTG+PM for our example is shown in Figure 1. At the left hand side, the FTG is shown which presents the different formalisms used throughout this paper (*e.g.*, Plant, Environment, ReachabilityGraph), as well as all operations between them (*e.g.*, Plant2EPN, Combine, Analyze, Mark). At the right hand side, the PM is shown which presents the order in which these operations are defined, as well as the specific modelling artefacts that are propagated between operations (*e.g.*, Combine combines the Encapsulated Petri nets which originate from the Plant, Control, and Environment, together with the architecture model). The flow of the problem at hand is easy to deduce: domain experts create models for the plant, control, environment, architecture, and safety query in a domain-specific language, following the requirements defined beforehand. The plant, control, and environment model are individually translated to Encapsulated Petri nets (*i.e.*, Petri nets with ports), which are merged together with the architecture model. The resulting Petri net is analyzed for reachability, on which the safety query is executed. If the query is found, an unsafe situation can be reached, and the offending series of instructions is presented to the user. Users are then prompted to make revisions to their models. If the query is not found, the system is deemed safe, and the process finishes.

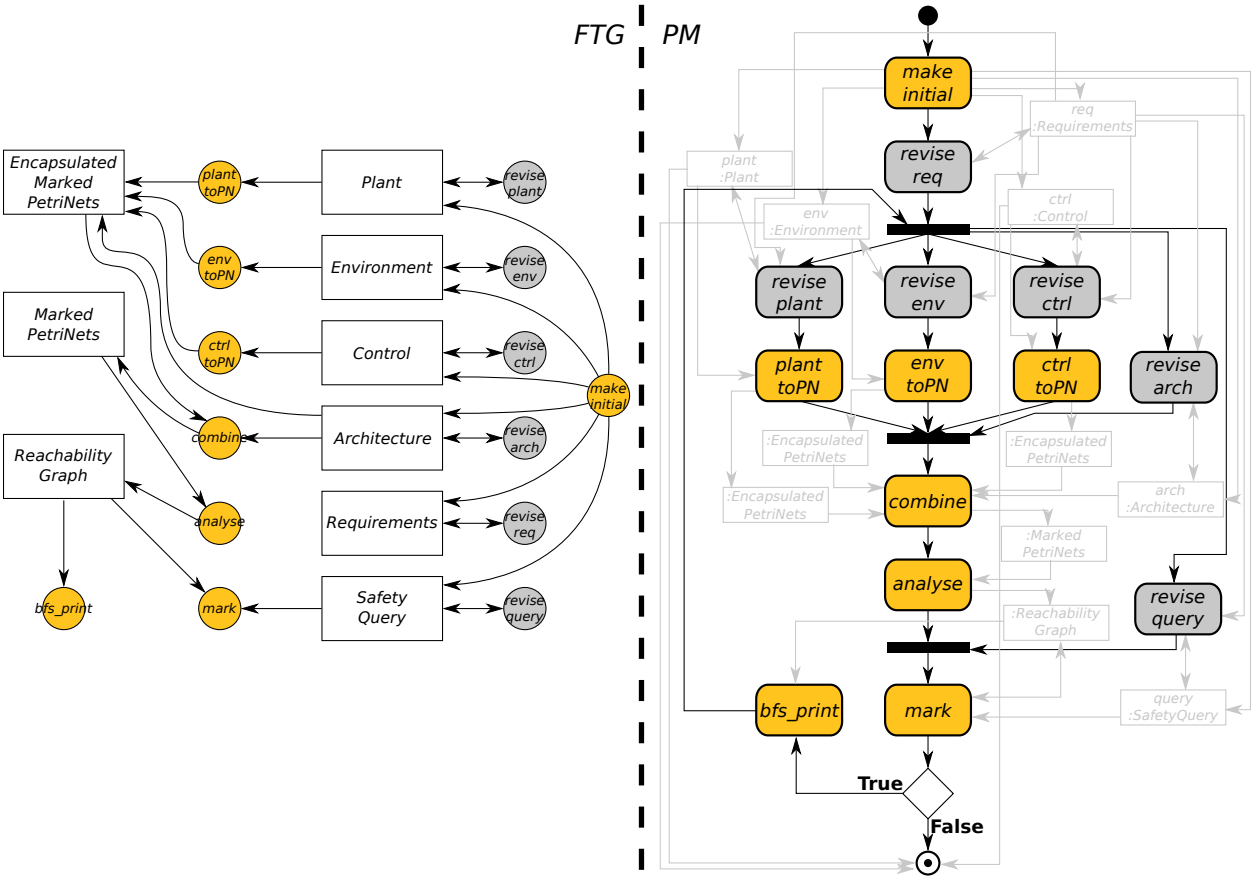

Figure 1: FTG+PM of our example: the development and verification of a power window.

3 ASPECT 1: LANGUAGE ENGINEERING

The first aspect of MPM is language engineering. Language engineering allows the creation and instantiation of new (modelling) languages. For example, SCCD (Van Mierlo et al. 2016) is such a new language which combines the existing languages for Class Diagrams and Statecharts. The primary reason for creating a new language, is to use it: SCCD models can now be created which combine models in the Class Diagrams and Statecharts languages. Users of that language are constrained by the language when modelling. For example, it doesn't make sense in SCCD to allow a connection between a class of Class Diagrams and a basic state of Statecharts. When the SCCD language was engineered, these decisions had to be made.

The language that was created must of course conform to another language: the meta-language. Many language engineering tools nowadays don't consider the creation of new languages too different from the creation of a model, as it is just an instance of the meta-language. Again, the meta-language is itself an instance of another language, and so on. To prevent infinite recursion, the topmost language is an instance of itself, which is termed meta-circularity. These different levels give rise to a meta-hierarchy, as shown in Figure 2 for the models and languages used in our example. At the topmost level, we see ClassDiagrams as the meta-circular level. Below, we see all formalisms used in the example, which were all created through language engineering. At the lowest level, all instances of the engineered languages are shown.

3.1 Relation to MPM

While General Purpose Languages (GPLs) can be used to tackle all problems, they often do so with huge accidental complexity. For example, it is possible to represent a power window controller directly in C code,

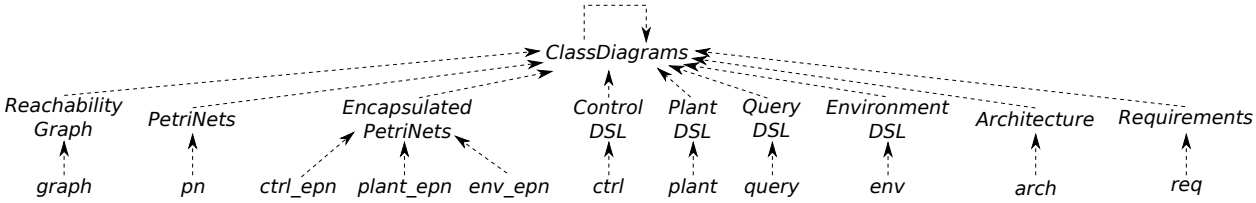

Figure 2: Meta-modelling hierarchy of the example.

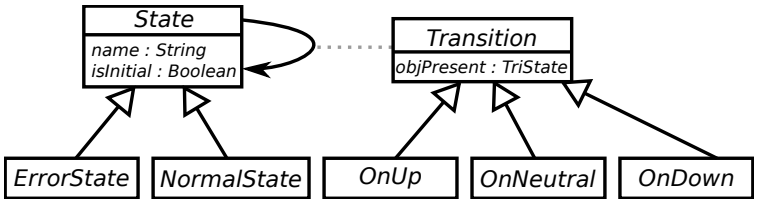

Figure 3: Plant metamodel, describing allowed constructs for a plant model.

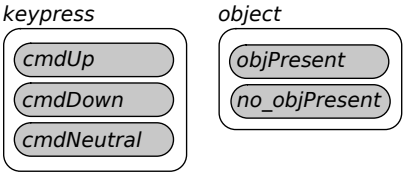

Figure 4: Environment model.

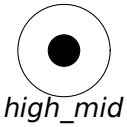

Figure 5: Safety query model.

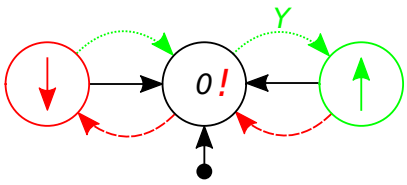

Figure 6: Control model.

though at great cost: all assumptions must be represented as well, significantly increasing verbosity and reducing the conceptual overview of the model. With language engineering, we gain the ability to easily create new languages specific to some domain: Domain-Specific (Modelling) Languages (DSMLs). These languages are a closer match to the problem domain, and can represent all assumptions at the language-level, instead of repeating them each and every time at the model-level. For example, a power window controller can be visualized as a state machine which responds only to three types of button press: up, down, and neutral. Some of the primary advantages of using DSMLs are decreasing verbosity, increasing conceptual clarity, and maximally constraining the modeller.

Given these advantages, it is clear why language engineering is relevant to MPM. As multiple domains are combined with the MPM approach, this implies different languages to be combined: one for each domain. These languages are domain-specific, and they are the only language the domain expert is confronted with. For example, the control engineer only uses a simple statemachine-like language, independent of all the underlying technology. This closes the conceptual gap between the problem domain and solution domain.

In an FTG+PM, as the one shown in Figure 1, languages are shown at the left-hand side, in the Formalism Transformation Graph (FTG). Each nodes represents a different language.

For the power window, we have implemented several domain-specific languages to help the development of the system. For example, the metamodel of the plant language is shown in Figure 3. It shows the different elements that can be used: two kinds of state, and three kinds of transitions.

These different domain-specific languages were subsequently used to model each part of the system. There is a model for the environment (Figure 4), safety query (Figure 5), control (Figure 6), plant (Figure 7), and architecture (Figure 8). These models are compact and provide a much more conceptually clear overview of the parts of the system, than if it were to be done in a GPL. For example, the control model would probably have taken several hundreds of lines of C code. Additionally, these models are closer to the problem they are modelling, and require no knowledge of programming or some GPL.

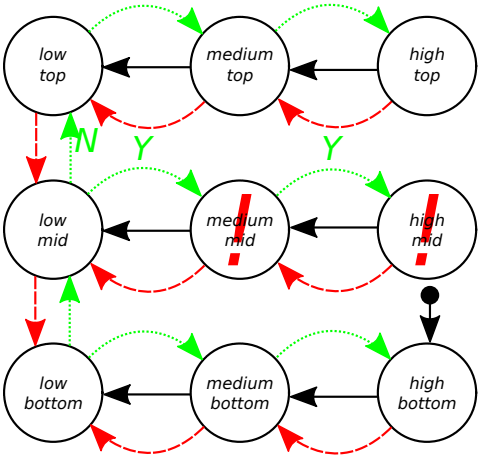

Figure 7: Plant model.

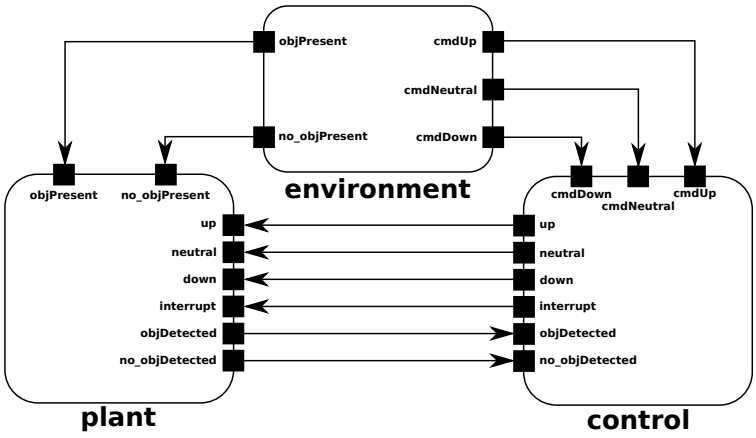

Figure 8: Architecture model.

3.2 Modelverse Support

The Modelverse allows for language engineering through meta-modelling: languages are themselves instances of the meta-language and are thus models themselves. This offers users the full possibilities to create new DSMLs, without modifying the tool. All models shown before have been successfully implemented. Note the inheritance between links in the plant metamodel: this is fully supported by the Modelverse, as associations can behave just like classes. For example, they can have attributes, subclasses, or even associations of their own. While this is an often desired feature, many tools fail to support it.

Supporting meta-modelling, and thus seeing languages as models as well, provides another advantage to the Modelverse. The Modelverse runs as a service and can share not only models, but languages as well. The Modelverse is therefore also a model repository. In combination with the MPM approach, this provides an additional benefit: languages can be reused (with minor changes). To protect intellectual property, sharing of models and languages can be disabled, or only allowed with a select group of users. This came in handy with the power window example, as the Petri nets, Encapsulated Petri nets, and Reachability Graph languages could be reused between different projects.

With the use of the Modelverse as a repository, many languages and associated models are created. To prevent the noise of all languages and models created by other users, the FTG side of the FTG+PM can be seen as a view on the complete set supported by the Modelverse. Only relevant languages and models are shown, based on the languages that are actually used (later on) in the process model, and can therefore change as the process is updated. The FTG helps as an overview of the used languages, and also makes sure that all used languages can be resolved.

4 ASPECT 2: MODEL OPERATIONS

The second aspect of MPM is model operations. Model operations define the semantics, or meaning, of models on which they operate. This is often called denotational or operational semantics, depending on how the semantics is defined. Denotational semantics give semantics by mapping the model to another semantically equivalent model, which is expressed in a language that has semantics. Operational semantics give semantics by executing the model directly, effectively changing the current model and creating an execution trace. Model operations can also be more general model management operations, such as model merging or splitting.

Model operations are often expressed using either (imperative) action code, such as EOL (Kolovos, Paige, and Polack 2006), or (declarative) model transformations, such as ETL (Kolovos, Paige, and Polack 2008). While we assume the reader to be familiar with imperative programming languages, an explanation of

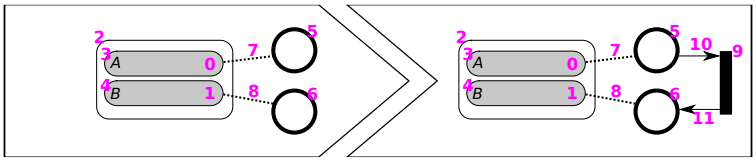

Figure 9: Rule for transforming the environment model to an encapsulated Petri net.

model transformation is provided. Model transformations are based on the work in the graph transformation community (Rozenberg 1997). Transformations are based on rewriting rules: the model is searched for the occurence of a specific pattern, after which it is rewritten. While there are many ways of visualization, we will use an explicit Left-Hand Side (LHS) and Right-Hand Side (RHS). In this representation, shown in Figure 9, the LHS represents the pattern to be found. When found, the pattern is replaced by the RHS. The purple labels relate elements in the LHS and RHS. Elements associated with labels that no longer occur in the RHS are removed, while elements with new labels are created when applying the rule. In this example rule, one step of the translation from environment model to encapsulated Petri net is shown.

4.1 Relation to MPM

The fact that MPM puts models in a central role makes model operations of great importance, as semantics for each DSL is required. Often, denotational semantics is the easiest to realise in the context of DSLs, as it allows DSLs to be mapped to more general modelling languages, for which semantics is defined. For example, all statemachine-like languages can be mapped to a generalized state machine language, for which semantics is defined. This is often easier than defining operational semantics for the DSL. With this denotational mapping, we can reuse operations defined on the target language. For example, Petri nets can both be simulated and analyzed, and therefore mapping to Petri nets for simulation automatically provides analysis functionality to the DSL.

Many approaches exist for model operations, though model transformation using RAMification (Kühne et al. 2009) is often used with MPM. With RAMification, transformation rules are modelled explicitly: a new language is created by Relaxing, Augmenting, and Modifying an existing language. As could be seen in Figure 9, the LHS and RHS seem to contain actual models in the DSL that is being operated on. Through RAMification, model operations can be seen as models themselves, offering all the usual modelling operations on model operations as well.

In the FTG+PM, model operations can be seen in the FTG side, where they constitute the circles connecting the different languages. Together with language engineering, this completes the FTG, creating an overview of the network of languages and operations used. Note the different colours of the operations in the FTG+PM: some operations are manual, whereas others are automatic. Sometimes operations need to be manual, and can therefore not be done automatically, as they require additional information to be added to the model. There can also be multiple input and output languages to a single transformation: if so, the models are merged or splitted before or after the actual operation.

In the power window example, we see a wide range of operations defined on the DSLs, most of them automated. For example, all DSLs have a mapping to the more general domain of encapsulated Petri nets. From this language on, we can continue through automatic operations. There are some manual operations involved as well: creating and revising DSL models happens manually. This is logical, as these operations are the creation (or correcting) of the models in the DSL, and therefore requires manual input.

4.2 Modelverse Support

The Modelverse supports the creation of model operations, created just like models: a model operation is a model itself. This means that there are several languages defined which allow their instances to be executed. Similar to model creation, adding or altering model operations does not modify the tool itself.

It is obvious in an MPM context that these too should be modelled using the most appropriate formalism. Though RAMification offers this for model transformations, not all problems are ideally expressed using model transformations. As discussed before, some tools use imperative languages, such as EOL or Java, whereas others use model transformations. Neither is better than the other for all problems: some problems are just better expressed using an imperative language, and others are better expressed declaratively. Most tools are limited to only a single model operation approach: either model transformations or an imperative action language. The Modelverse allows both: model transformations through RAMification and explicitly modelled action language. In the Modelverse, the action language is also modelled explicitly, meaning that its instances (*i.e.*, code) are modelled explicitly as well.

In our example, there are some operations that are ideally modelled using model transformations, as they heavily rely on matching, and others which are ideally mapped using an imperative action language. For example, model transformations are easy to map DSLs to the more general petri net representation, as we provide a mapping from patterns in the source language to constructs in the target language. An imperative action language, on the other hand, is ideal for operations that are well-studied and for which proven algorithms exist. For example, reachability analysis of a Petri net is well-studied and algorithms are readily available. Implementing this in code takes only about a hundred lines of code, whereas this would require many different model transformation rules, which need simple iterative (or recursive) algorithms.

Once more, the Modelverse offers additional advantages. As model operations are also seen as models, they can be shared between users, if permissions allow. Reusing model operations as well as models allows users full access to other formalisms by only defining one additional transformation for the DSL. For example, reachability analysis of Petri nets can be reused for multiple purposes and across multiple domains. Having this already implemented is a huge boost, as only the transformation from the DSL to Petri nets needs to be created.

5 ASPECT 3: PROCESS MODELLING

The third and final aspect of MPM is process modelling. When processes become big, it becomes difficult, if not impossible, to execute them in a reproducable way. An explicitly modelled process greatly helps, since it can be used for documentation, enactment, analysis, or even optimization.

A process model generally consists of two types of nodes: operations and data. These form the two types of flow in a process model: control flow and data flow.

Control flow originates at the start node and is followed throughout the execution. When the control flow encounters an operation, the operation is executed. Afterwards, control passes on to the next operation. There are some special operations, such as fork, join, and decision nodes. While we stick to a simple process model, process models can be augmented with allocation of operations to specific users, priority of operations, expected duration, and so on.

The data flow specifies which models are used by which operations. While control flow dictates which operations must execute, it doesn't state on which data it should be executed. Data flow manages the different modelling artefacts that are created throughout the process, and decides which data is passed on to which operation. Data both serves as input and output of the operation.

5.1 Relation to MPM

As presented up to now, MPM gives rise to many formalisms, models, and operations on them. Keeping an overview of the process, or how these many models are used, is therefore essential. Going from the initial requirements to the final safety check is non-trivial and can become difficult to reproduce if it is not documented, as we pass through many intermediate formalisms, each of which excels in one specific domain. This is the case for both control flow (which operations to execute), and data flow (on which data do the operations execute), as operations might be executed multiple times, but at different pieces of data.

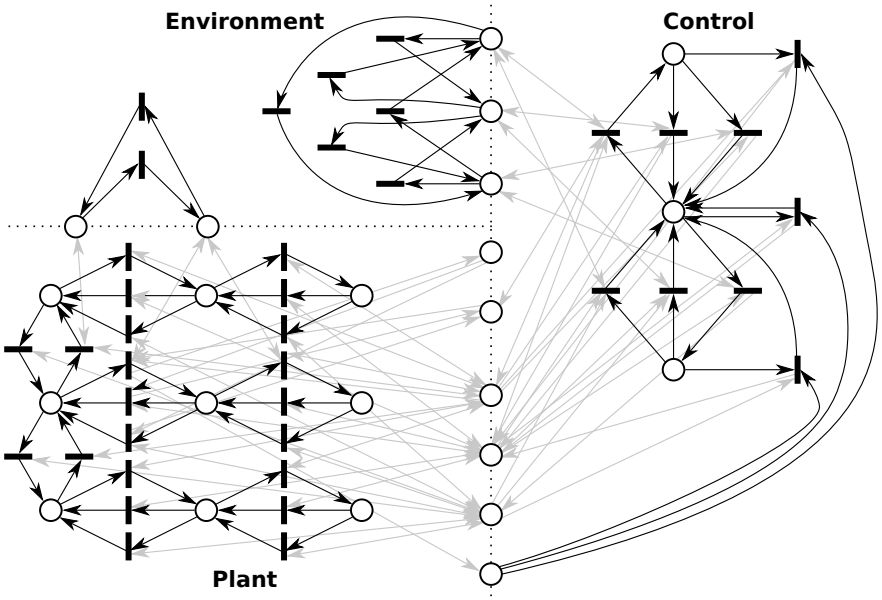

Figure 10: Combined Petri net, automatically generated from the DSL models.

An explicitly modelled process gives a complete overview of the control and data flow in the process. Ideally, the process model not only serves as documentation, but also for enactment support. With enactment, the process model is actually used for automated chaining of operations. Operations are executed in the correct order (or a correct order, if multiple possible orders exist), with the right data as input and output.

The process models themselves can also have input and output models. Input models refer to previously defined models in some other process. Output models are models that can be used later on, even in a different context. For example, our process model in the example could take the requirements document as input model, and return the different DSL models as output, while hiding the intermediate models completely. These DSL models can then be used in a different process, such as one that simulates these components, thereby allowing for modular composition of processes.

In the FTG+PM, the PM is obviously the process model. The process model is tightly connected to the FTG, as the PM refers to the languages and operations defined therein. It furthermore linearizes the FTG such that it is no longer just a graph of formalisms and relations: it becomes structured for our problem.

In our example, we had the different formalisms, models in these formalisms, and operations to map between different formalisms. There was, however, no order in which this should happen, and how data has to be utilized. This information is added when the PM is taken into account: it becomes clear where we must start, and which operations to execute in what order. Additionally, the PM shows the different artefacts created throughout the process. Many of these artefacts are not shown to the modellers at all, as they are intermediate models of automatic transformations. For example, the composed Petri net, shown in Figure 10, is never shown to the modellers, though it is generated and is the model that is analyzed. With the process model, we see that there are only a few manual operations, where domain experts must model their part of the system. Afterwards, they get feedback on whether the total model is safe or not.

5.2 Modelverse Support

The Modelverse supports the use of process models as discussed up to now, including the enactment of process models. Using enactment, the automatic operations are chained. Manual operations are partially automated: the correct models and formalisms are pre-loaded, and the user is presented with a prompt in which the required changes can be made.

The processes themselves are again models. Combined with the repository nature of the Modelverse, this makes it possible for processes to be shared between users, if permissions allow. Additionally, sharing the FTG+PM guarantees reproducability for other users, as they merely have to enact the same PM, instead of manually following a (potentially incomplete or ambiguous) script.

In combination with all other aspects of the Modelverse, we get full FTG+PM support in the Modelverse, including enactment. The FTG+PM from our example can easily be enacted and will prompt users for input when manual operations are required. Users are only prompted to create the different DSL models, while all intermediate models and operations remain hidden. After execution of the relevant operations, modellers are presented with the results of safety analysis. This results in either "safe" or a counter-example showing users how a violating state can be reached.

6 RELATED WORK

Existing tools support only one, or at most two, of the three domains supporting MPM, making it impossible to use a complete FTG+PM. We discuss related work for each aspects, and describe where they are lacking.

For language engineering, we consider the problems of meta-modelling and domain-specific modelling. Tools supporting this are often referred to as language workbenches. Examples are AToM^3 (de Lara and Vangheluwe 2002), AToMPM (Syriani et al. 2013), WebGME (Maróti et al. 2014), and MetaDepth (de Lara and Guerra 2010). All are specialized for language engineering, and allow users to create new languages and instantiate them further. Some tools have support for model operations as well, though only limited. For example, MetaDepth supports model transformations through ETL (Kolovos, Paige, and Polack 2008) and operations through Java (de Lara and Guerra 2010), whereas AToMPM uses RAMification (Kühne et al. 2009).

For model operations, we consider the problems of model management and executable models. Model management is focussed on generic model operations, and does therefore not go into detail about the semantics of individual models. One such example is MMINT (Di Sandro et al. 2015). It cannot be used to define new languages, but can be used to manage languages and define relations between them. Executable modelling attaches a meaning to existing models, often done through the use of model transformations. For example, Groove (Rensink 2004) allows for efficient graph transformations, but ignores most aspects of language engineering and process modelling. Note that some of the language engineering tools are capable of model operations, as presented in the previous paragraph. In contrast to the Modelverse, tools restrict themselves to a single way of expressing operations: either through model transformations or programming code. This makes it difficult to specify operations that are not an ideal match to that specific language.

For process modelling, we consider the actual modelling of the process, and the associated enactment. While most language engineering tools can be used to model the process, as it is possible to create new languages, there is no associated enactment support, as the model is without semantics. For example, AToMPM and MetaDepth can be used to create a process model, but it is limited to documentation only. In the domain of business process modelling, standards such as the BPMN allow for the description of a business process, and also supports some enactment. However, there is no support for the management of models (in domain-specific languages), and for the execution of operations such as model transformations.

In contrast to the Modelverse, neither of these tools provide support for flexible language engineering, model operations in the most appropratie formalism, and process modelling and enactment. While some attempts have been made in AToMPM to include process modelling and enactment (Mustafiz et al. 2012), this support ran into limitations of the framework.

In addition to these supported aspects, the Modelverse also serves as a repository, further strengthening the usefullness of each aspect. While other repositories exist, such as ReMoDD (France, Biemand, and Cheng 2006) and MDEForge (Basciani et al. 2014), they focus almost exclusively on the repository aspect, mostly ignoring MPM. While ReMoDD does not support model semantics, MDEForge allows users to upload ATL rules, though not as models.

Many applications to the power window example have already been done. The verification aspect of the power window has previously been described in detail, which formed the basis for our work and underlying models (Mosterman and Vangheluwe 2004). Our paper, however, goes deeper into each individual aspect, by proposing even more restricted DSLs, making the model operations more explicit, and providing a process model with enactment support. Additionaly, our paper verifies more error scenario's, by explicitly allowing multiple keypress interleavings, as well as multiple object insertion and removals. This significantly increases the resulting Petri net and resulting reachability graph. Support for the FTG+PM has been presented previously in the context of verification of the power window (Denil 2013), but did not include the of the Petri net. Additionally, our work provided tool support for the presented FTG+PM, actually giving it executable semantics and implementation support.

7 CONCLUSION

We have argued for the need for MPM to tackle the complexity of current systems. The importance of the FTG+PM in the context of multi-paradigm modelling was also highlighted.

We considered three aspects of multi-paradigm modelling: language engineering, model operations, and process modelling. With language engineering, domain-specific languages can be created to lower the gap between the problem and solution domain. With model operations, semantically meaningful relations between these different languages are constructed. With process modelling, the network of languages and operations on these languages becomes structured for a specific process.

While support exists for each of these individual aspects, their combination is not supported by current tools. We have presented the Modelverse as a tool with support for both language engineering, model operations, and process modelling and enactment. This makes the Modelverse capable of full support for MPM and the FTG+PM. These capabilities were exemplified through the use of the power window example. To this end, we have created several domain-specific languages for the power window, coupled with operations to map them to Petri nets, and perform reachability analysis on the generated Petri nets. The full FTG+PM was modelled and enacted in our tool. Th repository nature of the Modelverse further enabled the reuse of languages, models, model operations, and process models, if permissions allow.

In future work, we aim at making the Modelverse fully collaborative, possibly parallelizing process enactment over different, geographically distributed, users. While the Modelverse is a technical enabler of MPM, many open theoretical problems remain, such as behavioural equivalence of models in model operations when using abstraction and refinement.

ACKNOWLEDGMENTS

This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO) and was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

REFERENCES

Basciani, F., J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio. 2014. "MDEForge: an Extensible Web-based Modeling Platform". In *Proceedings of the Workshop on Model-Driven Engineering On and For the Cloud (CloudMDE)*, 66 – 75.

de Lara, J., and E. Guerra. 2010. "Deep Meta-modelling with MetaDepth". In *Proceedings of the TOOLS EUROPE Conference*, 1 – 20.

de Lara, J., and H. Vangheluwe. 2002. "AToM^3: A Tool for Multi-formalism and Meta-modelling". In *Fundamental Approaches to Software Engineering*, 174–188.

Denil, J. 2013. *Design, Verification and Deployment of Software-Intensive Systems: A Multi-Paradigm Modelling Approach*. Ph. D. thesis, University of Antwerp.

Di Sandro, A., R. Salay, M. Famelis, S. Kokaly, and M. Chechik. 2015. "MMINT: a Graphical Tool for Interactive Model Management". In *Proceedings of the MoDELS Demo and Poster Session*, 16 – 19.

France, R., J. Biemand, and B. H. C. Cheng. 2006. "Repository for Model Driven Development". In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 311 – 317.

Kolovos, D. S., R. F. Paige, and F. A. C. Polack. 2006. "The Epsilon Object Language (EOL)". In *Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications*, 128 – 142.

Kolovos, D. S., R. F. Paige, and F. A. C. Polack. 2008. "The Epsilon Transformation Language". In *Theory and Practice of Model Transformations*, 46 – 60.

Kühne, T., G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. 2009. "Explicit Transformation Modeling". In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 240 – 255.

Lucio, L., S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. 2013. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". In *SDL 2013: Model-Driven Dependability Engineering*, Volume 7916 of *Lecture Notes in Computer Science*, 182–202.

Maróti, M., R. Kereskényi, T. Kecskés, P. Völgyesi, and Ákos Lédeczi. 2014. "Online Collaborative Environment for Designing Complex Computational Systems". *Procedia Computer Science* 29 (0): 2432 – 2441. 2014 International Conference on Computational Science.

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

Mustafiz, S., J. Denil, L. Lúcio, and H. Vangheluwe. 2012. "The FTG+PM Framework for Multi-Paradigm Modelling: an Automotive Case Study". In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, 13–18.

Rensink, A. 2004. "The GROOVE Simulator: A Tool for State Space Generation". In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science, 479–485.

Rozenberg, G. (Ed.) 1997. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. River Edge, NJ, USA: World Scientific Publishing Co., Inc.

Syriani, E., H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. "AToMPM: A Web-based Modeling Environment". In *Proceedings of MODELS'13 Demonstration Session*, 21–25.

Van Mierlo, S., Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe. 2016. "SCCD: SCXML Extended with Class Diagrams". In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML*, 2:1–2:6.

Van Tendeloo, Y. 2015. "Foundations of a Multi-Paradigm Modelling Tool". In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*, 52 – 57.

Vangheluwe, H., J. de Lara, and P. J. Mosterman. 2002. "An Introduction to Multi-Paradigm Modelling and Simulation". In *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, 9 – 20.

YENTL VAN TENDELOO is a PhD student in the Modelling, Simulation and Design (MSDL) research Lab at the University of Antwerp (Belgium). In his Master's thesis, he worked on the MDSL's PythonPDEVS simulator, a simulator for Classic, Parallel, and Dynamic Structure DEVS. His PhD is on the conceptualization and development of the Modelverse, a new meta-modelling framework and model management system. His e-mail address is Yentl.VanTendeloo@uantwerpen.be.

HANS VANGHELUWE is a Professor at the University of Antwerp (Belgium), and an Adjunct Professor at McGill University (Canada). He heads the Modelling, Simulation and Design (MSDL) research lab distributed over both universities. In a variety of projects, often with industrial partners, he develops and applies the theory and techniques of Multi-Paradigm Modelling. Recently, he has been active in the design of Mechatronic and Automotive applications. He is the chairman of the European COST Action IC1404 "Multi-Paradigm Modelling for Cyber-Physical Systems". His e-mail address is Hans.Vangheluwe@uantwerpen.be.