Foundations of a Multi-Paradigm Modelling Tool

Yentl Van Tendeloo
University of Antwerp
Email: Yentl.VanTendeloo@uantwerpen.be

***Abstract*—None of the current plethora of meta-modelling tools includes a complete, explicit model of themselves. Such a model, a precise specification of the tool's syntax and semantics, allows for introspection and self-modifiability. These features enhance model debug-ability, make it easier to decompose the tool for distributed execution, and allow for reasoning about correctness and performance. In this paper, we present the foundations of the Modelverse, a self-modifiable environment for Multi-Paradigm Modelling (supporting multi-formalism and multi-abstraction modelling as well as explicitly modelled processes). We identify a set of requirements, which we believe are vital to a modern Multi-Paradigm Modelling tool. These requirements then are mapped to the features which our specification will support: self-modifiability, formalization, and multiple notions of conformance between models and meta-models.**

I. Introduction

To deal with the increasing complexity and size of the systems that we build, both physical, software and combinations thereof, Multi-Paradigm Modelling (MPM) [1] promotes the explicit modelling of all aspects of system development using the most appropriate abstraction(s) and the most appropriate formalism(s). It adresses and integrates three orthogonal research dimensions: *model abstraction*, concerned with the (refinement, generalization, ...) relationships between models at different levels of abstraction; *multi-formalism modelling*, concerned with the coupling of and transformation between models described in different formalisms; and the *explicit modelling* of the (multi-user, collaborative, multi-domain) model management processes.

A crucial feature is user collaboration, with every user possibly from a different domain, collaborating from different locations, at different times. This is usually adressed by presenting the modelling tool as a service, which ideally runs forever, requiring online self-updating to guarantee high availability.

Most current modelling tools do not directly support use as a service, or they do not allow online self-updating, through self-modification. Self-modification requires a description of itself, preferably in the form of a model, to allow explicitly modelling of the modification transformations. As such, a *complete* model of the modelling tool, and all of its features, needs to be present within the tool itself, and be expressed in the most appropriate formalism(s). Besides including the operational semantics of its execution, including model management operations, and thus allowing for self-modification, the execution context should also be included. This allows debugging through direct inspection of the execution data (or trace, if an appropriate trace model is used).

Collaboration between users with different expertise poses challenges for consistency management of shared models, due to the required links between the different models. Each of these models are possibly in different formalism, using multiple levels of abstraction, and have multiple simultaneous views. Such consistency links should also be modelled explicitly, which introduces the need for properly typing them.

Our contribution is the specification of the foundations of a self-modifiable, multi-paradigm modelling environment: the Modelverse. We start by eliciting the requirements we find essential to such an environment, resulting in three high-level features to be fulfilled by our specification, and thus by all implementations of the Modelverse. The presented foundations describe a class of Modelverse realizations, which satisfy our identified set of requirements.

We state our identified requirements, and our proposed features for such a tool in Section II. Section III then defines the semantics and both the interface of the tool and its components are defined. Section IV introduces the distinction between several conformance dimensions along with the notion of linguistic conformance. Section V presents related work. Conclusions are given in Section VI, which also presents how these features will be used in future work.

II. Requirements

We start by identifying a set of requirements for our foundations, which we deem to be important for any MPM tool. Where appropriate, we reference this requirement back to other tools which support this feature, or to the publication that explains its importance to MPM.

- **Forever running**: the Modelverse should always be able to continue running. As such, exceptional situtations should be dealt with. Examples are a crashing program due to division by 0 or to numerical overflow and non-terminating simulations. Also, modifications to the behaviour, for example to fix bugs, or to introduce new features, should not require a restart. An exception are changes to the minimal (static) kernel, which defines the action language semantics.
- **Model Everything Explicitly**: every element in the Modelverse needs to be explicitly modelled, using the most appropriate formalism, resulting in higher analyzability and formality. This expands to the execution of action code, for which the execution context should be explicitly modelled, increasing debugability. The importance of modelling every aspect of a system was identified in [1].
- **Multi-View**: the Modelverse should support different views on the same model, for example hiding or aggregating parts of a model to create a different view. Multi-view is frequently used when multiple users

are involved, coming from different domains, such as in [2].

- **Multi-Abstraction**: the Modelverse should be able to reason at different levels of abstraction. This is one of the requirements for an MPM tool [1].
- **Multi-Formalism**: the Modelverse should support combining multiple formalisms. This is one of the requirements for an MPM tool [1].
- **Multi-User**: the Modelverse should be able to serve multiple users simultaneously, guaranteeing some degree of fairness between different users. Multiple users are supported in two orthogonal dimensions: either through the support of multiple concurrent interfaces, and through multiple users for a single interface. It is one of the main features of WebGME [3], and is also supported in other tools such as like AToMPM [4].
- **Interoperability**: the Modelverse might have several implementations, though all should be able to interoperate, as they have to modify the same conceptual graph. WebGME [3] is again an example for this. In WebGME too, a distinction is made between different internal components.
- **FTG+PM**: the Modelverse should have an explicit FTG+PM [5]. First, it should contain an automatically generated Formalism Transformation Graph, describing the different formalisms present in the Modelverse, and the relationships (i.e., transformations) between them. Second, process Process Models should also be present in the Modelverse, to prescribe the behaviour of the use of the Modelverse. This is one of the requirements for an MPM tool [1].

To satisfy these requirements, we propose three main features of the Modelverse, which will be discussed in the remainder of this paper.

- **Formalisation**: all parts of the Modelverse should be formalized to establish the semantics of each component. This fulfills the following requirements:
  - *Interoperability* is achieved because all components will implement the same interface, with exactly the same semantics.
  - *Multi-User* is achieved because all interleaving semantics are formalized, resulting in deterministic behaviour, even in the presence of multiple users.
- **Self-modifiability**: the Modelverse should contain a model of itself, and should be able to alter it, resulting in changed behaviour. This fulfills the following requirements:
  - *Forever Running* is achieved because updates to the Modelverse can happen on-the-fly.
  - *Model Everything* is achieved because to be self-modifiable through model operations, the Modelverse should be modelled in itself.
- **Multi-conformance**: a single model should be able to conform to multiple meta-models simultaneously. This supports the following requirements:
  - *Multi-View* is supported by having each view as a different conformance relation, which can be coupled to a specific concrete syntax.

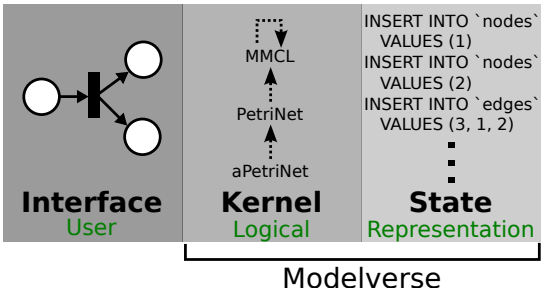

Fig. 1: Overview of the Modelverse architecture

  - *Multi-Abstraction* is supported because different metamodels can signify different levels of abstraction.
  - *Multi-Formalism* is supported through the application of different conformance relations for each part of the model seperately. Inter-formalism links are supported by conforming to a special physical conformance relation, which is closely tied to the Physical Type Model (see the next section).

III. The Modelverse

An architectural overview of the Modelverse is presented in Fig. 1. The Modelverse consists of two main components: the Modelverse State (MvS) and the Modelverse Kernel (MvK). They share a common concept to reason about the data they are manipulating, which is a conceptual graph representation. Different Modelverse Interfaces (MvI), capable of communication with the Modelverse, exist outside of the Modelverse.

In the MvI, the user is presented with a graphical or textual front-end to the Modelverse, matching the needs of that particular user. Several implementations of the MvI can be used concurrently. The MvI translates all user operations, in whatever way it accepts, to operations which the MvK understands. As this is the only interface to the end-user, the MvI is concerned with usability. The MvK considers models at the logical level, where it can reason about conformance relations and enforce them through syntax-directed editing. As we clearly distinguish between the MvK and the MvS, the MvK does not know how the model is physically represented. The MvK is therefore concerned with the semantics, and the execution of models. At the lowest level, the MvS receives operations on the conceptual graph, and maps them to the representational level, where it is actually stored. The MvS is solely concerned with the physical representation of the conceptual graph, and offers a uniform interface, independent of how it is stored internally.

We will first elaborate on the conceptual graph representation, which is used as the base representation of all models. Afterwards, we describe how the MvS maps models in this graph, to an actual implementation. Finally, the MvK defines the semantics of the models present in the MvS, by defining transformation rules that specify how the state evolves.

A. Conceptual graph

Conceptually, all data in the Modelverse is stored in the form of a graph, as defined below. This graph can hold a primitive value in a node, and both nodes and edges can be connected using edges. All elements (node or edge) can be

accessed using a unique identifier. An actual implementation can store the graph in different physical representations (*e.g.*, using a relational database or triplestore). This allows for more specialized implementations, depending on the problem domain, while still being interoperable.

We define a graph G, element of $\mathcal{G}$ (the set of all possible states of the Modelverse). A graph consists of nodes (N_G), possibly with values (in $\mathbb{U}$) defined on them ($N_{V,G}$), and edges (E_G, identifiers in $E_{IDS,G}$). Edges may connect both nodes and edges. Nodes and edges have a unique identifier, with IDS_G being the set of all identifiers. When constructing an edge, it is required that both the source and target already exist before the construction of the edge. This guarantees, by construction, that all edges are (eventually) rooted in nodes, and that no infinite recursion is possible when iterating over the graph.

$$G = \langle N_G, E_G, N_{V,G} \rangle \in \mathcal{G}$$
$$n_i \in N_G \subseteq IDS_G$$
$$e_j \in E_G \subseteq IDS_G \times IDS_G \times IDS_G$$
$$N_{V,G} : N_G \to \mathbb{U}$$
$$E_{IDS,G} = \{b | (a, b, c) \in E_G\}$$
$$N_G \cap E_{IDS,G} = \emptyset$$
$$N_G \cup E_{IDS,G} = IDS_G$$
$$\forall e_i, e_j \in E : e_i = (a, b, c), e_j = (d, e, f), (b = e) \Rightarrow (e_i = e_j)$$

With $\mathbb{U}$ defining the set of all possible types: $\mathbb{U} = \mathbb{I} \cup \mathbb{F} \cup \mathbb{S} \cup \mathbb{B} \cup \mathbb{A} \cup \Sigma_{type}$. We define the following primitive types, for which the MvS needs to provide native support: **Integer** ($\mathbb{I}$) as the set of integers; **Float** ($\mathbb{F}$) as the set of floating point numbers; **String** ($\mathbb{S}$) as the set of all ordered combinations of ASCII characters; **Boolean** ($\mathbb{B}$) as either True or False; **Action** ($\mathbb{A}$) as an action language construct, used to define the semantics later on; and **Type** (Σ_{type}) as the set of all supported types. As none of these sets overlap, it is possible to infer the type of the provided data. Note the distinction between $\mathbb{U}$ and Σ_{type}: $\mathbb{U}$ represents the type of data, whereas Σ_{type} represents a kind of meta-type of which all types are instances.

$$\mathbb{A} = \begin{array}{l}\{If, While, Assign, Call, Break,\\ Continue, Return, Input, Output, Define\}\end{array}$$
$$\forall i, j \in \{\mathbb{I}, \mathbb{F}, \mathbb{S}, \mathbb{B}, \mathbb{A}, \Sigma_{type}\} : i \neq j \Rightarrow i \cap j = \emptyset$$
$$\Sigma_{type} = \begin{array}{l}\{IntType, FloatType, StringType,\\ BooleanType, ActionType, TypeType\}\end{array}$$

B. Modelverse State

We now define the interface that all compliant modelling tools should implement. Our interface is defined on the previously defined graph, though the implementation will depend on the data structure used.

The MvS implements several atomic functions, which are to be used as the primitive Create, Read, Update, and Delete (CRUD) operations on the conceptual graph. Effectively, the MvS can be seen as a graph library, implementing an interface to our conceptual graph. Apart from defining only primitive CRUD operations, several composite operations are provided to allow for more efficient execution.

All CRUD operations are formalized as operations on the conceptual graph. A representative specification is the create edge operation C_{edge}, for which the semantics is defined next.

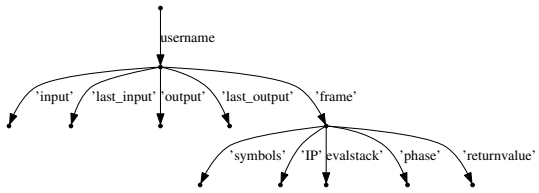

Fig. 2: Graph to match as execution context.

Informally, it extends the set of edges with a new edge, which gets an identifier that was not yet used.

$$C_{edge} : \mathcal{G} \times IDS \times IDS \to \mathcal{G} \times E_{IDS}$$
$$C_{edge}(G, i_1, i_2) = (G', i_3)$$
$$G = \langle N, E, N_V \rangle$$
$$G' = \langle N, E \cup \{e_i\}, N_V \rangle$$
$$e_i = (i_1, i_3, i_2) \notin E$$
$$i_3 \notin IDS_G$$

C. Modelverse Kernel

We will now consider the Modelverse Kernel (MvK), which is responsible for the execution of action code. As everything is modelled explicitly, the execution context will be part of the MvS. Graph transformations are used to define the semantics of our action language, which can be mapped to a series of MvS operations as previously defined. We use a concrete syntax which shows positive matches in solid black lines, negative matches in dotted red lines, delete matches in dashed blue lines, and create matches in thick green lines.

The well-formedness of the execution context can be checked using graph matching. Action code semantics are defined using graph transformations, transforming the execution context.

We define the execution context of the MvK, using graph matching. The execution context contains all execution data, which is required for the MvK to implement the semantics of arbitrary formalisms. If the number of matches is not exactly, the execution context is malformed and deterministic execution (which we currently require) is impossible.

A structure is matched as presented in Fig. 2. At the top of the structure sits the Modelverse root node, which is a known node. From this root node, there is a link to all user root nodes, containing the name of the user. From the user root, links point to the explicitly modelled input and output queue, which are used by the user to interact with the Modelverse. There is also a link to the currently active execution frame, which contains all data necessary for execution.

With the execution semantics and state completely defined in the Modelverse, the final remaining step is implementing operations in the action code itself. These operations are then able to perform modifications on the MvS, with as a special case modifying their own definition. As such, we achieve *self-modifiability*. Self-modifiability as required for debugging functionality, such as modifying variables, is supported by explicitly storing the execution context.

What remains for our formalization is the semantics of each of the action language constructs. For each construct, defined in $\mathbb{A}$, the required modifications of the execution context need to be defined using graph transformations.

We give the transformation rules for a While instruction

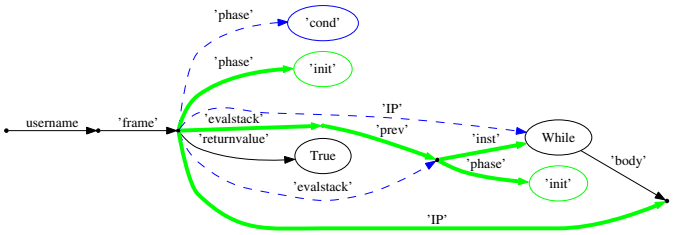

Fig. 3: Evaluate the body by moving the instruction pointer (IP) and pushing the while instruction back on the evaluation stack.

in Fig. 3, showing its semantics. The instruction pointer is moved to the body of the While construct, on the condition that the returnvalue is True. Reevaluation of the while construct is scheduled to happen when the body is completely executed, by putting this instruction on the evaluation stack.

These graph transformation rules are defined such that there should always be exactly one possible match. If no matches can be found, this indicates that the execution context, the current action language primitive, or both, are invalid. If multiple matches are found, non-determinism is possible, which is disallowed.

In the presence of multiple users, interleaving is necessary between them to guarantee fairness. This prevents uninterruptable loops at the lowest level, as each basic transformation rule is guaranteed to terminate.

Note that we do not define the semantics of model management operations at this level. As this is the static core of the Modelverse Kernel, its semantics is fixed and cannot be altered dynamically. Model management operations are implemented on top of this minimal layer. It is then possible to modify their semantics at runtime, to access their explicitly modelled semantics from within the Modelverse, and to support different versions simultaneously.

We have now achieved *formalisation*, as all action code models can now be given semantics by mapping them to operations on the MvS.

IV. Conformance

By allowing a single model to conform to multiple metamodels, using multiple distinct conformance relations, we can combine strict metamodelling with explicitly model management operations. The conceptual graph, representing the model, is thus interpreted depending on the metamodel/-conformance relation being used. Examples of metamodels are a problem-domain specific metamodel (*e.g.*, a Petri Net metamodel), and a more physically oriented metamodel (*e.g.*, a Graph metamodel). With the domain-specific metamodel, users can work using a domain-specific language, and are therefore maximally constrained (*i.e.*, syntax-directed editing). With the graph metamodel, users are unconstrained as there is no interpretation given to the graph (*i.e.*, free-hand editing). Even with the graph metamodel, however, the modelling environment still enforces conformance, but just with a very loose metamodel. Depending on the metamodel, different level hierarchies are constructed due to a different interpretation of the conceptual graph.

Fig. 4 presents different notions of conformance that can be devised using the Modelverse. The use of two different dimensions is based on the dimensions identified in [6]. While

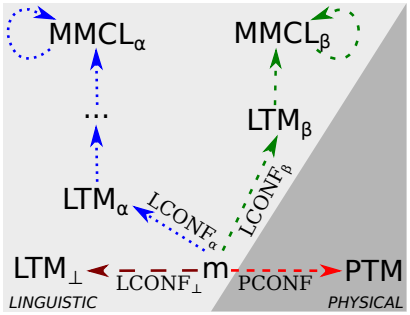

Fig. 4: Different conformance relations

the number of distinct conformance relations can vary, each model must have a (physical) conformance mapping to the Physical Type Model (PTM), to physically represent the model. It will also have a (linguistic) conformance mapping to the Linguistic Type Model (LTM), which is basically a metamodel of our conceptual graph, using conformance$_\perp$. Thanks to this additional conformance relation, to which everything conforms, the user gains access to the physical level in an explicitly modelled way.

A. *Conformance*$_\perp$

As all our data is (conceptually) represented using a graph, the graph instance can also be interpreted as a linguistic instance of a graph metamodel. Because all operations constrain the result to be a well-formed graph, all models in the Modelverse conform to this metamodel by construction. Since every model in the Modelverse is conceptually representable as a graph, everything can be flattened to a single level, conforming to the graph formalism. Within this level, all operations and links between elements are non-level crossing, and can therefore be correctly typed. Links that are normally level-crossing, or inter-formalism, can now be correctly typed without violating strict metamodelling constraints.

B. *Conformance*$_L$

Beyond this built-in conformance relation, which is always satisfied, users can define multiple linguistic conformance relations as well. Each of these conformance relations, given them satisfying the conditions mentioned by [7], induces meta-levels, which can vary according to the selected conformance relation. Linguistic conformance cannot be guaranteed at all, and requires checking whenever either the model or the metamodel change, to ensure it is enforced.

Because a conformance$_L$ view is only a specific view on a model, a single model can conform to multiple metamodels. A *conforms* function is defined to determine linguistic conformance.. It takes two graphs – a model and a metamodel, which are both subgraphs of the conceptual graph – and a mapping between them, returning a boolean whether this typing is valid or not. This mapping encapsulates all typing information, thus typing is completely separated from the model and metamodel, allowing for maximal flexibility.

This definition of the typing relation, of which there might be multiple, makes us achieve *multi-conformance*.

V. Related work

Current meta-modelling tools support our identified high-level requirements to a varying degree. We now explore the

current state of the art for the three different features, and how other approaches support these.

Most of the recent meta-modelling tools are not, or only partially, *formalized*. While most tools formalize their internal data structure and its operations, they frequently ignore the action language. Tools such as AToMPM [4] or JavaUML[8], simply re-use a general purpose programming language. While this enhances reusability of currently existing tools, therefore solving the problem of lacking tools [8], this often results in a mismatch between desired and offered functionality [9]. While it is debatable whether the language is formal or not, it is not explicitly modelled, making self-hosting difficult.

A plethora of popular action languages exist, such as txtUML [10], xMOF [11], EOL [12], or ALF [13]. They are not modelled explicitly at the required level. While they do offer formal semantics, instances are again not represented explicitly as models, but merely in the form of normal programming languages. Closest to our action language is Kermeta [9], where the action language is explicitly modelled and action code is also represented as a model to be executed. The execution context is not explicitly represented though, nor how the action language constructs modify it.

A common example of a *self-modifiability* programming environment is Squeak [14], a Smalltalk interpreter written in Smalltalk [15]. Here, self-modifiability allows the programming environment to update itself, modifying its behaviour. To obtain self-modifiability, one requirement is that a model of the currently executing program is present in itself, such that it can be modified. This model cannot simply be executed as-is though, as there still needs to be a mapping to the physical level. In Squeak, there are two options: either the interpreter is run within a running (binary) interpreter, or the interpreter is translated to a binary form. If the interpreter is itself interpreted, it can modify its behaviour at run-time, and detailed insight is given in the running interpreter. If the interpreter is translated to a binary form, the main advantage is that the level of abstraction is raised: the interpreter can be written in a high-level, interpreted language, instead of writing the interpreter in a low-level, compiled language. Both approaches can be combined, by first writing and debugging the interpreter in a known interpreter, as it is in Squeak, after which it is bootstrapped.

Similarly to Smalltalk, we present a minimal static core, with all more advanced operations built on top of that core. Everything, except for this static core, can be viewed as a normal program (model), and can be modified at runtime.

Self-modifiability is completely different from self-hosting or bootstrapping (*e.g.*, writing a C compiler in C, and later on compile it with itself), which is fairly common. For self-modifiability, it is required that the executing program has access to its own behaviour, and is able to change it at run-time. While not impossible for compiled programs, it is significantly easier for interpreted programs due to their increased introspection capabilities. Most interpreters, however, are not self-hosted due to the possibly significant performance overhead. Notable exceptions to this are Squeak [14] and PyPy [16], which are written in Smalltalk and Python, respectively.

The notion of *multi-conformance* is a middle ground between level-agnostic modelling [17] and strict metamodelling [7]. While we still comply with strict metamodelling, switching between different notions of conformance allows us to see everything at a single level, as a direct instance of the top level model, as done by XMF-Mosaic [18].

Multi-conformance is also used in the application of language relaxation [19], as required for transformations [20].

Enterprise Architecture Frameworks (EAF), such as the Zachman framework [21], make explicit the notion of multiple users, multiple views, distributed architecture, and interaction between different tools. As all artefacts are be supported in the Modelverse, instances of this framework can also be created. Ultimately, an EAF can be used for the Modelverse itself, during the process of bootstrapping.

While we technically allow multi-level modelling thanks to our conformance functions, it is not at the same level as more specialised tools such as Melanie [22] or MetaDepth [23], for example as we do not support potency [7] directly.

VI. Conclusion

We described the Modelverse, a self-modifiable multi-paradigm modelling tool. Several requirements were presented which served as our guideline while making decisions on the specification. Three high-level requirements were identified: *formalisation*, *self-modifiability*, and *multi-conformance*.

By mapping all operations, up to the highest level, back to operations on our conceptual graph, we achieved the *formalisation* requirement. Doing so, we allow for scalability (more efficient implementations can be combined), interoperability (components can be switched out), and multi-user (interleavings fully defined).

Self-modifiability was achieved through the explicit modelling of all operations in the provided action language itself. Additionally, the complete execution context was explicitly modelled and represented in the Modelverse itself. This allows us to satisfy the *forever running* (updates can happen on-the-fly) and *model everything* (explicitly modelled operations and execution) requirements.

Finally, we showed how multiple conformance relations can be valid at the same time, offering *multi-conformance*. This helps satisfy the *multi-view* (each view is just a different conformance relation) and *multi-formalism* (inter-formalism links are well-typed) requirements.

In future work, we plan to further exploit the power we gained through our three high-level requirements:

1) *Formalisation* allows for different implementations, with different characteristics, such as performance. This formalisation futher allows us to support parallel and distributed implementations.
2) *Self-modifiability* allows for *self-describability*, making it possible for the Modelverse to contain a model of itself. Such models can range from performance models, to models that can be used for code synthesis. Self-modifiability also increases debugability of the action language, as explicitly modelled debuggers [24] can be written, which are able to inspect and modify the execution stack of the code under study.
3) *Multi-conformance* allows the creation of links between different models and metamodels. Such functionality is of critical importance for operations relating to consistency management, where multiple models might have relations between them.

ACKNOWLEDGMENT

This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO).

REFERENCES

[1] P. J. Mosterman and H. Vangheluwe, "Computer Automated Multi-Paradigm Modeling: An Introduction," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 80, no. 9, pp. 433–450, 2004, special Issue: Grand Challenges for Modeling and Simulation.

[2] A. Finkelstein and H. Fuks, "Multiparty specification," *SIGSOFT Software Engineering Notes*, vol. 14, no. 3, pp. 185–195, Apr. 1989. [Online]. Available: http://doi.acm.org/10.1145/75200.75228

[3] M. Maróti, R. Kereskényi, T. Kecskés, P. Völgyesi, and Ákos Lédeczi, "Online Collaborative Environment for Designing Complex Computational Systems," *Procedia Computer Science*, vol. 29, no. 0, pp. 2432 – 2441, 2014, 2014 International Conference on Computational Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050914004049

[4] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, "AToMPM: A Web-based Modeling Environment," in *MODELS'13 Demonstrations*, 2013.

[5] L. Lucio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss, "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains," in *SDL 2013: Model-Driven Dependability Engineering*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7916, pp. 182–202. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38911-5_11

[6] C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," *ACM Trans. Model. Comput. Simul.*, vol. 12, no. 4, pp. 290–321, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/643120.643123

[7] T. Kühne, "Matters of (Meta-)Modeling," *Software and System Modeling*, vol. 5, pp. 369–385, 2006.

[8] P. Neubauer, T. Mayerhofer, and G. Kappel, "Towards integrating modeling and programming languages: The case of UML and Java," ser. Proceedings of the 2nd International Workshop on the Globalization of Modeling Languages, 2014, pp. 23–32.

[9] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving Executability into Object-oriented Meta-languages," in *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, ser. MoDELS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 264–278. [Online]. Available: http://dx.doi.org/10.1007/11557432_19

[10] G. Déva, G. F. Kovács, and A. Ancsin, "Textual, executable, translatable UML," ser. 14th International Workshop on OCL and Textual Modeling Applications and Case Studies, 2014, pp. 3–12.

[11] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xMOF: Executable DSMLs Based on fUML," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, M. Erwig, R. Paige, and E. Van Wyk, Eds. Springer International Publishing, 2013, vol. 8225, pp. 56–75.

[12] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon Object Language (EOL)," in *Model Driven Architecture Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Rensink and J. Warmer, Eds. Springer Berlin Heidelberg, 2006, vol. 4066, pp. 128–142. [Online]. Available: http://dx.doi.org/10.1007/11787044_11

[13] "OMG ALF," http://www.omg.org/spec/ALF/, 2013.

[14] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '97. New York, NY, USA: ACM, 1997, pp. 318–326. [Online]. Available: http://doi.acm.org/10.1145/263698.263754

[15] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[16] A. Rigo and S. Pedroni, "PyPy's Approach to Virtual Machine Construction," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 944–953. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176753

[17] B. Henderson-Sellers, T. Clark, and C. Gonzalez-Perez, "On the search for a Level-Agnostic Modelling Language," *Lecture Notes in Computer Science*, vol. 7908, pp. 240–255, 2013.

[18] T. Clark, C. Gonzalez-Perez, and B. Henderson-Sellers, "A Foundation for Multi-Level Modelling," in *MULTI 2014 Multi-Level Modelling Workshop Proceedings*, 2014, pp. 43–52.

[19] R. Salay and M. Chechik, "Supporting agility in MDE through modeling language relaxation," in *Proceedings of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2013).*, 2013, pp. 20–27.

[20] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Explicit transformation modeling," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 6002, pp. 240–255.

[21] J. Zachman, "A framework for information systems architecture," *IBM Systems Journal*, vol. 26, no. 3, pp. 276–292, 1987.

[22] C. Atkinson and R. Gerbig, "Melanie: Multi-level Modeling and Ontology Engineering Environment," in *Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards*, ser. MW '12. Innsbruck, Austria: ACM, 2012, pp. 7:1–7:2. [Online]. Available: http://doi.acm.org/10.1145/2448076.2448083

[23] J. de Lara and E. Guerra, "Deep Meta-Modelling with MetaDepth," in *Proceedings of TOOLS, Lecture Notes in Computer Science vol. 6141*. Springer, 2010, pp. 1–20.

[24] S. Van Mierlo, Y. Van Tendeloo, B. Barroca, S. Mustafiz, and H. Vangheluwe, "Explicit modelling of a Parallel DEVS experimentation environment," in *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, ser. DEVS '15. Society for Computer Simulation International, 2015, pp. 860–867.