Modelverse specification

Yentl Van Tendeloo
Bruno Barroca
Simon Van Mierlo
Hans Vangheluwe

March 14, 2016

Abstract

None of the current plethora of (meta-)modelling tools include a complete model of themselves. Such a model, a precise specification of the tool's syntax and semantics, allows for introspection and reflection. This enables features such as debugging. Without such a model, it is harder to decompose a tool into components for distribution, reason about efficiency, and reuse components of existing implementations. In this technical report, we present the foundations of the Modelverse, a self-describable environment for multi-paradigm modelling (supporting multi-formalism and multi-abstraction modelling and explicitly modelled processes). The foundations describe a class of Modelverse realizations, which satisfy our identified set of requirements. Conceptually, all information in the Modelverse is stored in a graph, and model management operations transform this graph. Parts of the graph also describes action constructs which, amongst others, can be used to define (linguistic) conformance relations, the basis for multi-layer multi-level modelling.

Contents

1 Introduction **4**
1.1 Architecture 4

2 Related work **6**
2.1 Self-describable 6
2.2 Action language 7
2.3 (Meta-)Modelling tools 8

3 Axioms **9**
3.1 Axiom I: Forever Running 9
3.2 Axiom II: Scalability 9
3.3 Axiom III: Minimal Content 10
3.4 Axiom IV: Model Everything 10
3.5 Axiom V: Human Interaction 10
3.6 Axiom VI: Test-Driven 10
3.7 Axiom VII: Multi-View 10
3.8 Axiom VIII: Multi-Formalism 11
3.9 Axiom IX: Multi-Abstraction 11
3.10 Axiom X: Multi-User 11
3.11 Axiom XI: Interoperability 11
3.12 Interconnections 11

4 Modelverse State **13**
4.1 Data representation 13
4.2 CRUD interface 15

5 Modelverse Kernel **20**
5.1 Graph transformations 20
5.2 Execution context 23
5.3 Execution primitives 24
5.4 Primitive operations 39
5.5 Interface 41

6 Modelverse Interface **43**
6.1 Browser 43
6.2 Serialized Graph 43
6.3 HUTN for Action Language 43
6.4 HUTN for Models 43

7 Network communication **45**

7.1 Motivation 45
7.2 Modelverse State 45
7.3 Modelverse Kernel 45
7.4 Modelverse Interface 47

8 Conformance **49**
8.1 Graph conformance 50
8.2 Linguistic conformance 50
8.3 MMCL 51

9 Practical information **55**
9.1 Requirements 55
9.2 Test suite 55
9.3 Running the Modelverse 55
9.4 Bootstrap file 55
9.5 XML/HTTP requests 56
9.6 Compiling with HUTN 56
9.7 Examples 56

10 Conclusion **59**

1 Introduction

To deal with the increasing complexity and size of the systems, both physical and software and combinations thereof, that we build, Multi-Paradigm Modelling (MPM) [1] promotes the explicit modelling of all aspects of system development. It addresses 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.

User collaboration is usually solved by presenting the modelling tool as a service, which ideally would be constantly running, requiring online self-updating to guarantee high availability.

Most current modelling tools do not directly support usage 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 for explicit 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. Besides including the operational semantics of its execution, including model management operations, and thus allowing for introspection, reflection, and self-modification, we should also include the execution context. Thus allowing debugability through direct inspection of the execution data.

In the presence of multiple users, possibly collaborating from different locations, interoperability between the modelling tools is required. Therefore, these tools need to agree on a common exchange format, with precisely defined semantics. As such, all models need to be representable using this common exchange format, as done by most other tools.

Collaboration between users with different expertise raises challenges for consistency management of shared models, which would require links between the different models. Each of these models possibly in different formalism, using multiple levels of abstraction, and with multiple simultaneous views.

Our contribution in this paper, is the specification of the foundations of a self-describable, multi-paradigm modelling environment: the Modelverse [2]. We start by eliciting the requirements we find essential to such an environment, which will later be used in our specification. The presented foundations therefore describe a class of Modelverse realizations, which satisfy our identified set of requirements. Conceptually, all information in the Modelverse is stored in a single graph, and model management operations transform this graph. Parts of the graph also describe action constructs which, amongst others, can be used to define several linguistic conformance relations.

1.1 Architecture

An architectural overview of the Modelverse is presented in Fig. 1.1. The Modelverse consists of two main components: the Modelverse State (MvS) and the Modelverse Kernel (MvK), with a communication layer in between. Different Modelverse Interfaces (MvI), capable of communication with the Modelverse, exist outside of the Modelverse.

In the Modelverse Interface, the user has a graphical or textual front-end for the Modelverse, which is close to the problem domain. The MvI translates all user operations to operations for the MvK to process. The MvK considers models at the logical level, where it can reason about conformance relations and can enforce syntax-directed editing. For communication with the MvS, a conceptual idea of what a model looks like physically is used: the Physical Type Model (PTM). As we clearly distinguish

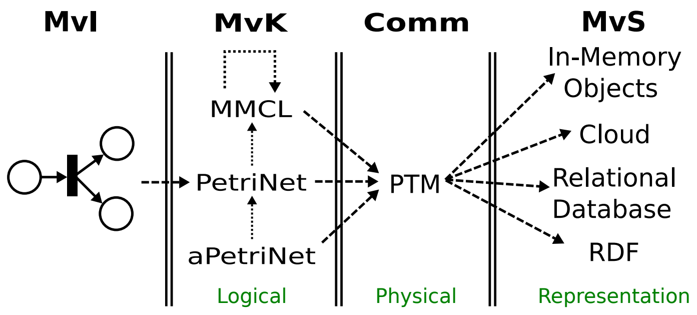


Figure 1.1: Overview of the Modelverse architecture

between the MvK and the MvS, the MvK cannot know how the model is represented in hardware. The PTM is therefore used as a common concept to reason about. Finally, the MvS receives changes on this PTM and maps them to the representational level, where it is actually stored in hardware.

2 Related work

Our specification builds on the work from several topics. We identify the following topics: self-describable environments, action languages, and (meta-)modelling tools. We touch upon all of these, as we create a modelling tool, which can describe itself using an action language.

In this chapter, we will go over these different topics, and relate back to our work.

2.1 Self-describable

The idea of having a self-describable programming language is not new. Squeak [3], for example, is a Smalltalk interpreter written in Smalltalk. For execution, there are two options: either the Smalltalk code is executed from within a running interpreter, or the interpreter is translated to a different programming language, which can be compiled. The former approach offers self-modifiability at run-time, whereas the latter offers increased performance. The translator only works on a subset of Smalltalk though, which closely maps to C (the target language of the translation). It is possible to combine both approaches: first write and debug using the interpreter-in-interpreter approach, and afterwards translate that same code to a different programming language. The combination is actually the main motivation for their work: it is possible to create code in Smalltalk, a high-level language, and debug it with all provided tools, and afterwards generate C, a low-level language, and profit from its performance.

A similar project is PyPy [4], a Python interpreter written in Python. The Python code that is used for the definition is again a subset of valid Python code, called RPython, which again gets mapped to C. Their main motivation is performance: by raising the level of abstraction from C to RPython, it becomes far easier to implement advanced interpreters (*e.g.*, including a JIT compiler).

Going over to the modelling world, there is already some work done on a self-describable meta-modelling environment. One such tools is XMF [5], which is a self-describable kernel for multi-level modelling. Its kernel conforms to itself, though it cannot modify itself due to the lack of side-effects to functions. While it offers a foundation for multi-level modelling, it does not include an explicitly modelled action language.

In the Modelverse, we also want to explicitly model the MvK component. But as the MvK can hold all kinds of models, the MvK should be able to manipulate (another instance of) itself. This would mean that the MvK is explicitly modelled in the Modelverse, and therefore is present in the MvS. As such, an instance of the MvK can modify its own model in the same way as this was possible with Squeak.

All tools provide an extensive model library, which is written in the language they implement. This is certainly the case for Smalltalk, where basically everything is a library, even primitive data structures such as integers. Tools can provide a minimal foundation, possibly hardcoded, with all other parts building upon this core. For example in PyPy, most libraries are implemented in pure Python code, whereas in CPython (the reference implementation), these are coded in C for efficiency reasons. The Modelverse is similar in this aspect, as it only has a minimal base of hardcoded operations, to which everything should be mapped. While this limits performance (since some operations require multiple primitive instructions), the amount of fixed components is minimized and thus the amount of modifiable code is maximized. Furthermore, it offers users insight into the working of the tool, without resorting to the kernel manual. Finally, new implementations of the kernel only require the reimplementation of the minimal core, with the complete library being reused as-is.

2.2 Action language

Several action languages are already in use today, to allow models with associated behaviour. We describe some of these languages:

- fUML [6] (foundational UML) is a frequently used language, often in combination with alf [7]. It is referenced as the assembly language of MDA [8], as many other languages get mapped to it. Work has been done to extend the library of fUML with external services, such as Java libraries [9, 10]. It is a quite elaborate language though, making it difficult to create a minimal core out of it. In its turn, this decreases the power of self-modifiability: a lot of constructs are built-in, and therefore their semantics cannot be changed.
- txtUML [11] (textual executable translatable UML) provides a mere interface for the user, where all code gets translated to Java. This has the advantage that existing tools, such as Java debuggers, can be reused. However, the question is as to whether existing general purpose debuggers are at the right level of abstraction, as they do not provide tracability information. Mapping to Java has the additional problem that it is not completely platform independent, as there would be a need for a Java interpreter at the platform. Additionally, Java is not explicitly modelled and therefore we would not be able to model everything explicitly.
- EOL [12] (Epsilon Object Language) is a language commonly used for model management purposes. Several closely related languages are also specialized for specific purposes, such as ECL (Epsilon Comparison Language), EML (Epsilon Merging Language), and EGL (Epsilon Generation Language). While it has wide support, and is based on the familiar Object Constraint Language (OCL), there is a lack of a complete metamodel. As the code is purely textual, and not explicitly modelled, it is impossible to combine with strict metamodelling, where level-crossing links (or accesses) are disallowed. Again, the language in itself is fairly large, making it difficult to implement a simple, minimal core.
- Kermeta [13] probably comes closest to the Modelverse. Their action language is explicitly modelled, and there is the possibility for both graphical and textual editting of action language constructs. However, they are limited in the use of strict, static typing and strict metamodelling, which is a potential problem in combination with model management operations.

We can identify two different design goals for action languages: operational semantics (examples in [14]), and model management operations (examples in [12]). Both goals have very different requirements. For the modelling of operational semantics, it is desired that the action language is also explicitly modelled, and therefore can directly reference the accessed values. Furthermore, we want to have requirements that are similar to strict metamodelling, where the action language is only able to access elements at the same level. It is therefore not possible to use such languages for the definition of model management operations: language constructs can only access a single level. For the modelling of model management operations, most approaches take some distance from the model itself, causing the language to not be modelled explicitly. This allows them to ignore strict metamodelling requirements, and access arbitrary elements from the model, even if they are at different levels. While this allows for model management operations, it is not possible to model the language, and therefore also impossible to use model management operations on themself, breaking self-describability and self-modifiability. While the definition of operational semantics is not impossible, there are no constraints on the modifications done to the model, which is a potentially dangerous operation. Due to this discrepancy, both of which we want to support in the Modelverse, there is a need for two different views: one specifically designed to define operational semantics, and another one to define model management and other core functions.

Languages also come in different representations, with purely textual being a popular choice for action language due to the similarity with programming languages. For the combination of behaviour and structure, however, we want a hybrid of these approaches: textual for the action language, but graphical for the structural metamodelling language [15]. As the action language model is also modelled explicitly, it is automatically possible to modify action language constructs with the graphical environment. The Modelverse takes the same approach as [15, 13, 9], as all tools are purely graphical internally, but with a textual front-end that will generate the appropriate models automatically.

In terms of debugability, many tools at the moment support some kind of execution traces [16], which can aid in debuging. Some, however, implement this through the use of an API that is exposed by the virtual machine that executes the code [17]. While this is a possible way, it makes clear that the virtual machine holds parts of the state that are not present in the state that is stored. Implying that the execution state is not explicitly modelled, and therefore not user-modifiable or debugable. It furthermore also raises questions as to whether the provided API is sufficient for all goals the debugger tries to solve. In the case of the Modelverse, no state whatsoever is stored in the virtual machine (the MvK), and everything is written to the MvS. Debuggers can therefore simply access and modify the complete execution state as if it were a regular model. Work based on traces, such as multi-dimensional trace files [18], can therefore be built on top of our provided execution state.

Finally, there is the formalisation aspect. Most languages are formalised by mapping to either fUML or Java. While this is a viable approach, this makes us dependent on the target language, reducing interoperability. An approach that lies closer to our approach in the Modelverse, is the mapping to graph transformations, as is done by [19]. As the MvS is conceptually a graph, a mapping of behaviour to graph transformations seems ideal.

2.3 (Meta-)Modelling tools

The last aspect to look at, is the functionality of current meta-modelling tools, in relation to the Modelverse.

WebGME [20] is a web-based tool, focussed on collaboration between multiple users. It really focusses on the collaboration aspect by offering model-based version control. Support for prototype-based inheritance distinguishes it from other tools. While it is a powerful metamodelling tool, it does not support model transformations. It is furthermore not possible to add in model transformations, as there is no minimal core on which transformations can be built. The Modelverse doesn't support model transformations at the moment, either, but these will later be implemented on top of the minimal core that is present. The architectural split between behaviour and storage was replicated in the Modelverse. In contrast, the behavioural part, the MvK, will be completely modelled in the action language defined by the Modelverse.

AToMPM [21] is another web-based meta-modelling tool with support for model transformations. While there is a notion of collaboration, this is not to the same degree as WebGME. The Modelverse has no direct notion of collaboration build-in, as such operations need to be implemented on top of the minimal core. The Modelverse doesn't include a default interface at the moment, as it only consists of the service.

Melanie [22] is a graphical multi-level modelling tool. It has support for automatic emendation (changes to a meta-level are automatically propagated to the lower levels), and transformations. While it comes with a graphical environment, there is no support for multiple users.

MetaDepth [23] is another multi-level modelling tool. It is purely textual, and therefore feels more familiar to programmers. While it lacks transformations, constraints and operational semantics can be defined through the use of action language in the EOL formalism. The Modelverse supports multi-level modelling by default, but mainly because there is no restriction on re-instantiation of previously instantiated elements. Advanced features, like emendation, or even potency, are not supported by default. Thanks to the minimal core, however, potency, or similar techniques, can be added in, as was done by XMF-Mosaic [5].

XMF-Mosaic [5] is one of the only self-describable multi-level meta-modelling tools. It is purely textual and there is no support for multiple users. However, it is also based on a minimal kernel, which is self-describable. The action language they use, EOL, is not self-describable though. Internally, the Modelverse makes fairly similar decisions, as for the use of a single common metamodel. While this is the only way of looking at the model in XMF-Mosaic, the Modelverse allows different conformance dimensions, of which one might actually be a *real* conformance dimension.

Kermeta [13] is similar to the Modelverse. It offers a hybrid syntax of graphical and textual, and also has a meta-model for the used action code. There is, however, a strong focus on strict and static typing, which, combined with strict metamodelling, can become very constraining for model management operations. Kermeta models many constructs explicitly: there is even a metamodel for primitive datatypes. These primitive datatypes however, just come out of a (system) library, and are therefore not easily changable, or visible, by the user.

While all mentioned tools have their specific application (and research) domain, the Modelverse tries to combine features of all tools. This is done through the use of the minimal core, on which the features of other tools can be built.

3
Axioms

We define a set of requirements for a Modelverse. These requirements, or axioms, will be used during our formalization to motivate our decisions. Although implementation-related requirements are not needed for our formalization, they are mentioned as it is something every implementation should conform to.

After an explanation of what each axiom represents, we give an overview of how all these axioms are related to each other.

3.1 Axiom I: Forever Running

The Modelverse should always be able to continue running. As such, no modifications to the behaviour should require a restart, except for changes to the (minimal) kernel (and thus the action language semantics). An (authorized) user should be able to alter all core concepts, with changes automatically applied for all connected Modelverse Interfaces.

Forever running also implies that the Modelverse runs as a service, separate from the MvI program, which is used by the user, but also on a different machine. A more drastic interpretation is that it should be parallelized and distributed, as to cope with possible hardware failure. We do not require this more drastic interpretation, though it is certainly a feature to take into account in an implementation evaluation.

The forever running does not apply to the MvI, of course, as the MvI is a tool ran on the system of the end-user. It is the whole of MvK and MvS that should run as if it is running forever.

3.2 Axiom II: Scalability

The Modelverse should be scalable in terms of computation, memory, number of users, number of models, and the size of individual models. Related to the previous axiom, scalability should still be maintained even if the Modelverse is forever running. Combined with scalability is performance: even if operations are scalable in terms of complexity, the total time taken by execution should also be as low as possible.

Due to our split in multiple components, we can also split up our scalability requirements over these components:

- The MvI needs to be scalable in performance, of course, though the size of models will be relatively small compared to those processed by the MvK or MvS, because the models being worked on will always be submodels of the *complete* Modelverse model. More important for the MvI is the scalability in the size of the model for visualization and presentation. Depending on the domain, an implementation might provide further methods for abstraction of components.
- The MvK needs to be scalable in performance, again, but mainly in the processing of action code constructs. An MvK instance should be easily parallelizable up to the “1 MvK per user” threshold. Beyond that limit, multiple MvKs would have to cooperatively work on a single block of action code, which is likely to hamper performance. An MvK also needs to be scalable in the number of users it is able to handle.
- The MvS needs to be scalable in performance, mainly in terms of the size of the complete Modelverse state. It is non-trivial to distribute or parallelise, as operations are small and atomic, and all data needs to be shared between users. The MvS should therefore be offloaded as much as possible, shifting all computation to the MvK. This reduces the functionality of the MvS to that of a simple, but high-performance, data structure library. Again, it should be scalable in the number of,

possibly simultaneous, requests made, which differs from the total number of users.

3.3 Axiom III: Minimal Content

A minimal amount of content should be available in the Modelverse by default. The content consists of the models necessary for bootstrapping, but also some default formalisms, such as Petri Nets, Parallel DEVS, Statecharts, FTG+PM [24], ...

For bootstrapping, the Modelverse contains a model of itself, which can then be compiled to a binary, executable outside of the Modelverse, or interpreted by the currently running MvK. From this viewpoint, the Modelverse will be similar to Squeak [3], which is a Smalltalk interpreter written in Smalltalk.

Apart from formalisms, some models should also be present in the Modelverse. These include the Formalism Transformation Graph (FTG), and the corresponding Process Model (PM), forming the FTG+PM. The FTG model can be automatically constructed from the formalisms that are automatically detected in the Modelverse. Combined with detecting the formalisms, it should also be possible to automatically detect all transformations defined between these formalisms, thus completing the FTG. The PM model will be the driving force of the MvK and defines which operations to execute. It can therefore be written in an action language, which defines the behaviour of the MvK, and thus the communication with the user.

3.4 Axiom IV: Model Everything

Every element in the Modelverse needs to be explicitly modelled, using the most appropriate formalism. This does not only include the typical elements, such as the models and metamodels, but should also go down to the level of the primitives such as Integer and Float. This will allow for stronger model transformations, as they can transform (and access) literally everything.

Ultimately, a model of the Modelverse should also be present in the Modelverse, which closes the loop. In the end, a compiled version needs to be used for pragmatic reasons, though this compiled version can be (automatically) compiled from the model that lives in the Modelverse.

Features like debugging, introspection, reflection, and self-modifiability will come from this axiom, as every part of execution is accessible for both reading and writing.

3.5 Axiom V: Human Interaction

All interaction with the human user of the Modelverse needs to be explicitly modelled. This includes timed behaviour of the Modelverse (*e.g.*, time-out of requests), or even the complete communication protocol. It is actually the MvI which will communicate with the Modelverse, though it will be guided by the user.

It should also be taken into account that the MvK will be (mainly) used by humans, and as such should be usable. While most of this will be handled by the MvI, which provides the tool to the user, the fact that a human is behind all of it should be taken into account. Possible applications for this are for performance evaluation: a human user has completely different (and likely slower) access patterns than an automated tool. The predefined constructs and design of the system should also be usable by humans, specifically those that are non-experts in design of the Modelverse. Enforcing strict metamodelling is part of the solution, as this offers users (and tools) a limited scope to worry about [**?**].

3.6 Axiom VI: Test-Driven

Development on the Modelverse should happen using the model of the Modelverse, which can be simulated, and placed in a variety of circumstances which are hard to replicate in real-life situations. A similar approach was taken by [25], where a DEVS model was made of a distributed DEVS simulation kernel. Modelling allowed them to replicate, among others, sudden disconnects, high latency connections, or different network topologies. Furthermore, detailed, and perfectly deterministic, performance insights can be gained by the simulation of the model. Certainly for parallel execution, this gives us deterministic thread interleavings, which can be crucial to debugging and performance analysis.

Functionality also needs to be checked as exhaustively as possible. Certainly for the first axiom, critical bugs should be avoided as much as possible. Because the Modelverse will have to communicate with a variety of tools, its interface will also have to be tested for conformance with the specifications.

3.7 Axiom VII: Multi-View

The Modelverse should support different views on the same model. Examples include hiding parts of a model, or aggregating different elements into a composite element. This gives rise to consistency management, as changes in one view will have to be propagated to all other views.

Multi-view should be handled at all components, as each component needs to allow it. The MvI needs to provide operations to use the different views, the MvK needs to update the views and keep them consistent, and the MvS needs to provide these operations efficiently. The MvS is least concerned with multi-view, as it sits at a lower level.

3.8 Axiom VIII: Multi-Formalism

The Modelverse should support models which combine different formalisms. Models should therefore be able to have a metamodel which is the combination of multiple (meta)models. Inter-formalism links should also be possible, even if those cannot be typed within the respective formalism. While the semantics of such a link depends on the domain, and therefore has to be provided by the user, the Modelverse should allow such links to be created and used. Consequently, links between models should also be possible, which can then act as the type for those inter-formalism links.

Related, a single model should be able to have multiple metamodels. A model could therefore be typed by a metamodel, but would also have to conform to a bigger metamodel, which contains the original metamodel as one of its elements. This allows the reuse of models, even if the context surrounding the metamodel has changed.

3.9 Axiom IX: Multi-Abstraction

The Modelverse should support systems which are expressed using a set of models, all at a different level of abstraction. Consistency management will again have to be handled here.

As was the case for multi-view, each component needs to think about multi-abstraction separately. The exception is again the MvS, as it is at a lower level. However, it can still (internally) use optimizations, knowing that some requests will be related to multi-abstraction.

3.10 Axiom X: Multi-User

The Modelverse should be able to serve multiple Modelverse Interfaces simultaneously. A main concern to this is fairness between users: a user cannot wait for its turn infinitely long. If a single user therefore uses all computational power, at the expense of other users, the code executed by this user will have to be automatically paused, marked as “low priority”, or terminated.

User Access Control is related to this, as users should be able to configure the Read/Write/Execute status of their models. As such, groups of users, with specific privileges, should also be supported.

If their access control allows it, users should also be able to read the state of the execution of other users. This will allow for debugging with multiple users: user *A* can execute code, with user *B* being an automated debugging bot, which examines the state of user *A*.

3.11 Axiom XI: Interoperability

Different implementations of the Modelverse and its interface should be possible. These implementations should all be able to communicate with each other, as long as they follow the same specification. This is one of our main goals for specifying the interfaces between components.

Additionally, because the semantics of action code and its corresponding execution context is defined, different MvK’s should be able to continue each other’s execution, or interpret the execution context of other tools. This can come in handy with different tools (*e.g.*, a debugger, a compiler, or an interpreter) which might be developed independently, though are able to understand each other’s information.

3.12 Interconnections

All of these axioms are related in some way, as the graph in Figure 3.1 shows. We now continue by explaining the links between all concepts, using their label:

1. As the Modelverse will be forever running, there is a need for garbage collection or periodical maintenance to guarantee a decent performance.

2. Having everything explicitly modelled allows us to create a self-modifiable Modelverse, which helps us with the forever running axiom.

3. In the presence of multiple users, it is necessary to have the Modelverse running as a service, which implies that it should run forever.

4. Using the performance tests, combined with the MvK being modelled explicitly, it becomes possible to assess the scalability of the Modelverse algorithms under specific workloads.

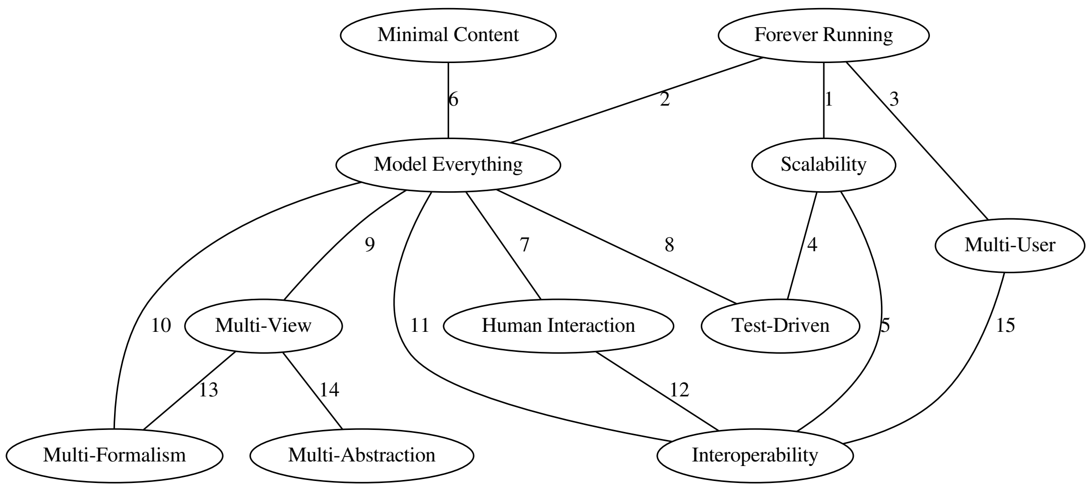


Figure 3.1: Overview of relations between all axioms

5. Scalability is deeply connected with interoperability, as there is often a trade-off: increasing interoperability will decrease scalability and vice versa.
6. Having everything modelled explicitly requires the presence of at least a few basic formalisms. Ultimately, it also includes having a model of the Modelverse in the minimal content of the Modelverse.
7. By modelling everything, we will inevitably also have to model the interaction with the human.
8. The performance tests will use a performance model of the Modelverse, which is contained in the Modelverse. To that end, the Modelverse will simulate its own performance.
9. Multi-view requires the ability to model everything, as we will have to model all different views separately.
10. By modelling everything explicitly, we also need to model links between different formalisms, which is a requirement for multi-formalism models.
11. Interoperability between different Modelverse components becomes easier if each component is modelled explicitly, as it clearly defines the expected semantics.
12. Interoperability is an essential part of human interaction, as otherwise it would be impossible for both of them to communicate.
13. Multi-view and multi-formalism are related due to a view being possibly expressed in a different formalism.
14. Multi-view and multi-abstraction are related, as different views might be at different levels of abstraction.

4
Modelverse State

We start our specification with the Modelverse State (MvS). The MvS maps the Physical Type Model (PGM) to the hardware. Essentially, the MvS needs to implement the CRUD interface using whatever algorithms and data representation it sees fit. Despite the liberal choice of data representation and algorithm, the interface is strictly defined and uses a special kind of graph, defined in this chapter. We will first describe the conceptual representation of the PTM, followed by the operations on it that the MvS should support.

4.1 Data representation

Conceptually, all data in the MvS is stored in the form of a kind of graph, as defined below. Informally, we define a graph which can have a primitive value in a node, and both nodes and edges can be connected using edges. Allowing edges to connect other edges allows for a more explicit representation, such as type links on associations, (Axiom IV: Model Everything). While edges between edges could also be conceptualized using the standard notions of graphs, using tricks similar to hypergraphs, having a closer mapping between the PTM and the models will allow for higher performance (Axiom II: Scalability). Both nodes and edges can be accessed using a unique identifier.

An actual implementation of this interface might store the graph in different physical representations (*e.g.*, using a relational database or triplestores). This allows for more specialized implementations, depending on the problem domain (Axiom II: Scalability), while still being interoperable (Axiom XI: Interoperability). Despite the need for multi-view (Axiom VII: Multi-View), multi-formalism (Axiom VIII: Multi-Formalism), and multi-abstraction (Axiom IX: Multi-Abstraction), everything is represented uniformly at this level. It is only at the Modelverse Kernel (MvK) level, that an interpretation is given to this graph.

We define a graph G, element of $\mathcal{G}$ (the set of all possible states of the MvS). A graph consists of nodes (N_G), possibly with values (in $\mathbb{U}$) defined on them (mapping $N_{V,G}$), and edges (stored in E_G as triples). Edges can run between both nodes and edges. All identifiers allocated to edges are stored in $E_{IDS,G}$. Nodes and edges have a unique identifier, with IDS_G being (exactly) the set of all identifiers. This also means that identifiers cannot be reused between nodes and edges.

Edges which are self-connecting can be problematic for certain recursive algorithms, which traverse an edge by going on to the source and target. Therefore, edges can, by construction, only link elements that already exist. This effectively prevents (indirect) links to itself. With this restriction, such constructs are disallowed and these recursive algorithms are therefore guaranteed to terminate. Such a restriction is also not limiting, as it is a normal requirement to only connect elements that already exist.

The requirement for ever increasing identifiers might seem contradictory to Axiom I: Forever Running, as the identifier would go up to infinity, consequently endangering Axiom II: Scalability. In theory, this is not a problem, though implementations will specifically have to handle this to prevent problematic situations (*e.g.*, integer overflow or slow operations). In an implementation, this could easily be solved by periodical *"identifier compaction"* (identifiers are reassigned, to filter out removed identifiers), or reusing removed identifiers (keeping in mind the constraint).

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

$\mathbb{U}$ defines the set of all possible values, or the union 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, supported in the PTM, for which the MvS needs to provide native support:

- **Integer** ($\mathbb{I}$) as the set of integers in the range $[-(2^{63}), 2^{63} - 1]$ (*i.e.*, as would be available using 64-bit integers);
- **Float** ($\mathbb{F}$) as the set of floating point numbers, as defined by IEEE 754, with double precision (*i.e.*, as would be available using 64-bit floating point numbers). Values outside of this range will be rounded towards $-\infty$ or ∞;
- **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 ($\{If, While, Assign, Call, Break, Continue, Return, Resolve, Access\}$), used by the Modelverse Kernel to define the semantics.
- **Type** (Σ_{type}) as the set of all supported types ($\{IntType, FloatType, StringType, BooleanType, ActionType, TypeType\}$). These types differ from the set denoting all elements (*i.e.*, $\mathbb{I}$ does not equal *IntType*), as the type is an instance itself.

We use $\mathbb{I}$ and $\mathbb{F}$, instead of $\mathbb{Z}$ and $\mathbb{R}$, respectively, because an implementation of these infinite concepts would not be able to exploit current hardware. This is required for Axiom II: Scalability, as otherwise primitive operations would be inefficient due to their generality. With this restriction, we enforce the size of the data values, thus preventing implementation-dependent behaviour (*e.g.*, some implementation using 32-bit integers, whereas another uses 64-bit integers).

The use of primitives does not violate Axiom IV: Model Everything, as primitives will still be explicitly modelled in the linguistic dimension. In the physical dimension however, we shift the representation of the data to the physical level to obtain higher performance (Axiom II: Scalability) and to have a basic type system available.

None of the value sets overlap, therefore it is possible to infer the type of the data, using N_T.

$$
\begin{gathered}
N_T : \mathbb{U} \to \Sigma_{type} \\
N_T(d) = \begin{cases} IntType & if \quad d \in \mathbb{I} \\ FloatType & if \quad d \in \mathbb{F} \\ StringType & if \quad d \in \mathbb{S} \\ BooleanType & if \quad d \in \mathbb{B} \\ ActionType & if \quad d \in \mathbb{A} \\ TypeType & if \quad d \in \Sigma_{type} \end{cases} \\
\forall i, j \in \{\mathbb{I}, \mathbb{F}, \mathbb{S}, \mathbb{B}, \mathbb{A}, \Sigma_{type}\} : i \neq j \Rightarrow i \cap j = \emptyset
\end{gathered}
$$

We can define a subgraph (M) of a graph (G), as a graph containing a subset of the nodes and edges, with the restriction that all used nodes and node values are copied. It is implicit that the resulting graph should still be valid according to the restrictions placed on the graph (*e.g.*, source and target of nodes is still present).

$$
\begin{gathered}
M \subseteq G \\
\Leftrightarrow \\
N_M \subseteq N_G \wedge \\
E_M \subseteq E_G \wedge \\
N_{V,M} \subseteq N_{V,G} \wedge \\
\forall (a \to b) \in N_{V,G} : a \in N_M \Rightarrow (a \to b) \in N_{V,M} \wedge \\
\forall (a,b,c) \in E_M : a, c \in IDS_M
\end{gathered}
$$

4.2 CRUD interface

The final part of the PTM is the interface, or the set of its supported operations, which are defined here. An MvS implementation needs to offer the operations defined here, irrespective of its implementation algorithm or data structure. Of course, the implementation does not need to be using a graph similar to the conceptual representation of the PTM, but the operations should always return exactly the same result.

We distinguish four different kinds of operations in our interface: Create, Read, Update, and Delete (CRUD). For each set of operations, we define the function signature and the required semantics.

Apart from the actual return value, operations also return a status. This status is an integer specifying a status code: $S = \mathbb{I}$. We have different categories of status codes: $1xx$ for success; $2xx$ for interface errors; and $3xx$ for execution errors. An interface error indicates an error in the call, for example wrong type of arguments. An execution error means that the call itself was well-formed, but could not be executed due to another restriction, such as an element not being defined.

All possible status codes are defined. Some additional errors might happen in the MvS though, such as out-of-memory problems. These errors are platform-dependent and are only caused due to the implementation, the hardware, or the combination of both. As such, an MvS is not allowed to return such errors and needs to handle such situations gracefully. For example, in the case of an out-of-memory error, the MvS needs to be able to swap out pieces of itself to disk, or over the network.

4.2.1 Create

The first set of instructions that we define are *create* operations. Create operations cause the creation of new elements in the graph, thus extending its size. Each newly created element will be assigned an identifier by the MvS, which is returned. It is this identifier which acts as the handle to that element in the MvS.

Note that there are no restrictions on the created identifier, apart from it being a value that is not yet used for another element. This allows whatever kind of identifier to be used, and even reuse is possible if the previous element was deleted.

First is the create node operation (C_N), which takes no arguments and returns the identifier of the newly created node, which was unused up to now.

$$\begin{gathered}
C_N : \mathcal{G} \to \mathcal{G} \times N \times S \\
C_N(G) = (G', n, 100) \\
G = \langle N, E, N_V \rangle \\
G' = \langle N', E, N_V \rangle \\
N' = N \cup \{n\} \\
n \notin IDS
\end{gathered}$$

The create edge operation (C_E) takes the identifier of the source and target elements (either a node or an edge) as argument, and returns the identifier of the newly created edge.

$$\begin{gathered}
C_E : \mathcal{G} \times IDS \times IDS \to \mathcal{G} \times E_{IDS} \times S \\
C_E(G, i_1, i_2) = (G', i_3, s) \\
G = \langle N, E, N_V \rangle \\
G' = \langle N, E', N_V \rangle \\
E' = E \cup \{e_i\} \\
e_i \notin E \\
e_i = (i_1, i_3, i_2) \\
i_3 \notin IDS \\
s \neq 100 \Leftrightarrow i_3 = None \\
s = \begin{cases} 200 & if \quad i_1 \notin IDS \\ 201 & if \quad i_2 \notin IDS \\ 100 & else \end{cases}
\end{gathered}$$

The last primitive create operation (C_{NV}) creates a new node, and assigns it a value immediately. It has the same signature as the create node, but takes a primitive value to assign to the created node. This operation could be implemented by first creating an empty node and afterwards updating its value, though this would negatively impact performance (Axiom II: Scalability).

$$
\begin{gathered}
C_{NV} : \mathcal{G} \times \mathbb{U} \to \mathcal{G} \times N \times S \\
C_{NV}(G,d) = (G',i,s) \\
G = \langle N,E,N_V \rangle \\
G' = \langle N',E,N'_V \rangle \\
N' = N \cup \{i\} \\
N'_V = N_V \cup (i \to d) \\
i \notin N \\
s = \begin{cases} 202 & if \quad d \notin \mathbb{U} \\ 100 & if \quad else \end{cases}
\end{gathered}
$$

For performance, we add a composite create operation, which creates a named edge between two graph elements (C_D). This operation is equivalent to creating an edge between the two elements, followed by creating an edge from the newly created edge, to the data value that was specified. It is formalised as follows.

$$
\begin{gathered}
C_D : \mathcal{G} \times IDS \times \mathbb{U} \times IDS \to \mathcal{G} \times S \\
C_D(G,a,d,b) = (G',s) \\
G = \langle N,E,N_V \rangle \\
G' = \langle N',E',N'_V \rangle \\
N' = N \cup \{c\} \\
E' = E \cup \{(a,i_1,b),(i_1,i_2,c)\} \\
N'_V = N_V \cup \{(c \to d)\} \\
c,i_1,i_2 \notin IDS \\
s = \begin{cases} 203 & if \quad a \notin IDS \\ 204 & if \quad d \notin \mathbb{U} \\ 205 & if \quad b \notin IDS \\ 100 & if \quad else \end{cases}
\end{gathered}
$$

4.2.2 Read

The next set of operations consists of the read operations. As there is no useful information in non-data nodes, there is no read operation defined on nodes, except for their primitive data (R_V). It is an error if the node that is being read does not have a value assigned to it.

$$
\begin{gathered}
R_V : \mathcal{G} \times N \to \mathbb{U} \times S \\
R_V(G,n) = (d,s) \\
G = \langle N,E,N_V \rangle \\
d = N_V(n) \\
s = \begin{cases} 206 & if \quad n \notin N \\ 300 & if \quad n \notin dom(N_V) \\ 100 & else \end{cases}
\end{gathered}
$$

Instead of a read operation on the nodes, it is possible to read out their outgoing edges (R_O) and incoming edges (R_I). This works for nodes, but also for edges, as edges can also be the source (and target) of other edges. The result is the identifier of the connected edges, in an unordered collection.

$$
\begin{gathered}
R_O : \mathcal{G} \times IDS \to 2^E \times S \\
R_O(G,i) = (e,s) \\
G = \langle N,E,N_V \rangle \\
e = \{(i,b,c) \in E\} \\
s = \begin{cases} 207 & if \quad i \notin IDS \\ 100 & if \quad else \end{cases}
\end{gathered}
$$

$$
\begin{gathered}
R_I : \mathcal{G} \times IDS \rightarrow 2^E \times S \\
R_I(G, i) = (e, s) \\
G = \langle N, E, N_V \rangle \\
e = \{(a, b, i) \in E\} \\
s = \begin{cases} 208 & if \quad i \notin IDS \\ 100 & if \quad else \end{cases}
\end{gathered}
$$

A read operation for edges (R_E) is defined as returning a tuple consisting of the source and target of the edge. Due to the restriction on the edge identifier, both the source and target identifier will be smaller than the edge identifier.

$$
\begin{gathered}
R_E : \mathcal{G} \times E_{IDS} \rightarrow IDS \times IDS \times S \\
R_E(G, i_1) = (i_2, i_3, s) \\
G = \langle N, E, N_V \rangle \\
e = (i_2, i_1, i_3) \in E \\
s = \begin{cases} 209 & if \quad i_1 \notin E_{IDS} \\ 100 & if \quad else \end{cases}
\end{gathered}
$$

For efficiency (Axiom II: Scalability), an additional *"dictionary read"* operation (R_{dict}) is defined to read an element which is linked to another one through an edge, which is connected to a node with a primitive value. This allows for a more efficient implementation of lookups from a specific node, without requiring an exhaustive search of the connected edges. While the search might still be necessary internally, implementations are free to create specialized data structures for this operation. Even if that is not the case, this operation reduces the amount of calls required to 1. Two errors are possible: if the specified entry could not be found in the dictionary, and if a matched link also has other outgoing links, causing ambiguity as to which element is the key to use.

Notice that there is room for ambiguity if a node has multiple outgoing links, linking to the same data value. While this could cause an error, we explicitly allow for this situation for performance reasons, as otherwise the search would always need to traverse all links, even if a match was already found.

$$
\begin{gathered}
R_{dict} : \mathcal{G} \times IDS \times \mathbb{U} \rightarrow IDS \times S \\
R_{dict}(G, i_1, v) = (i_2, s) \\
G = \langle N, E, N_V \rangle \\
d = N_V(v) \\
\exists b, c \in E_{IDS} : (i_1, b, i_2), (b, c, d) \in E \\
s = \begin{cases} 210 & if \quad i_1 \notin IDS \\ 211 & if \quad v \notin \mathbb{U} \\ 301 & if \quad \nexists b, c \in E_{IDS} : (i_1, b, i_2), (b, c, d) \in E \\ 302 & if \quad \exists b, c, e, f \in E_{IDS} : (i_1, b, i_2), (b, c, d), (b, f, e) \in E \wedge c \neq f \\ 100 & else \end{cases}
\end{gathered}
$$

Some other, more complex read operations on dictionaries are also supported, purely for efficiency reasons. Their errors are similar to the R_{dict} operation. Each of these operations returns a slightly different result, determined by the frequently used operations in the next section. These operations are:

- R_{dict_node} returns the element being linked to, but instead of a primitive value, it searches for a specific element by identifier. It therefore does not try to dereference the value stored in the resulting element, nor will it match different elements with the same value.
- R_{dict_edge} is equivalent as R_{dict}, but returns the identifier of the edge between them, instead of the element itself.
- $R_{dict\_reverse}$ returns a list of all elements that have an outgoing link towards the passed element, with the provided name on that edge. It is therefore basically a reverse dictionary lookup: return the dictionaries that contain this exact value with a specified key.

Multiple combinations would also be possible, though we only formalize those that are used by the MvK in later sections.

$$R_{dict\_keys} : \mathcal{G} \times IDS \rightarrow 2^{IDS} \times S$$
$$R_{dict\_keys}(G,a) = (l,s)$$
$$G = \langle N, E, N_V \rangle$$
$$\forall b,c,d,e \in IDS : (a,b,c),(b,d,e) \in E : e \in l$$
$$s = \begin{cases} 222 & if \quad i_1 \notin IDS \\ 100 & else \end{cases}$$

$$R_{dict\_node} : \mathcal{G} \times IDS \times IDS \rightarrow IDS \times S$$
$$R_{dict\_node}(G,i_1,i_2) = (i_3,s)$$
$$G = \langle N, E, N_V \rangle$$
$$\exists b,c \in E_{IDS} : (i_1,b,i_3),(b,c,i_2) \in E$$
$$s = \begin{cases} 212 & if \quad i_1 \notin IDS \\ 213 & if \quad i_2 \notin IDS \\ 303 & if \quad \nexists b,c \in E_{IDS} : (i_1,b,i_3),(b,c,i_2) \in E \\ 304 & if \quad \exists b,c,e,f \in E_{IDS} : (i_1,b,i_3),(b,c,i_2),(b,e,f) \in E \wedge c \neq f \\ 100 & else \end{cases}$$

$$R_{dict\_edge} : \mathcal{G} \times IDS \times \mathbb{U} \rightarrow IDS \times S$$
$$R_{dict\_edge}(G,i_1,v) = (i_2,s)$$
$$G = \langle N, E, N_V \rangle$$
$$d = N_V(v)$$
$$\exists b,c \in E_{IDS} : (i_1,i_2,b),(i_2,c,d) \in E$$
$$s = \begin{cases} 214 & if \quad i_1 \notin IDS \\ 215 & if \quad v \notin \mathbb{U} \\ 305 & if \quad \nexists b,c \in E_{IDS} : (i_1,i_2,b),(i_2,c,d) \in E \\ 306 & if \quad \exists b,c,e,f \in E_{IDS} : (i_1,i_2,b),(i_2,c,d),(i_2,f,e) \in E \wedge c \neq f \\ 100 & else \end{cases}$$

$$R_{dict\_node\_edge} : \mathcal{G} \times IDS \times IDS \rightarrow IDS \times S$$
$$R_{dict\_node\_edge}(G,i_1,i_2) = (b,s)$$
$$G = \langle N, E, N_V \rangle$$
$$\exists b,c \in E_{IDS} : (i_1,b,i_3),(b,c,i_2) \in E$$
$$s = \begin{cases} 216 & if \quad i_1 \notin IDS \\ 217 & if \quad i_2 \notin IDS \\ 307 & if \quad \nexists b,c \in E_{IDS} : (i_1,b,i_3),(b,c,i_2) \in E \\ 308 & if \quad \exists b,c,e,f \in E_{IDS} : (i_1,b,i_3),(b,c,i_2),(b,e,f) \in E \wedge c \neq f \\ 100 & else \end{cases}$$

$$R_{dict\_reverse} : \mathcal{G} \times IDS \times \mathbb{U} \rightarrow 2^{IDS} \times S$$
$$R_{dict\_}(G,i_1,v) = (l,s)$$
$$G = \langle N, E, N_V \rangle$$
$$d = N_V(v)$$
$$l = \{i_2 : \exists b \in E_{IDS}.(i_2,b,i_1),(b,c,d) \in E\}$$
$$s = \begin{cases} 218 & if \quad i_1 \notin IDS \\ 219 & if \quad v \notin \mathbb{U} \\ 309 & if \quad \nexists b,c \in E_{IDS} : (i_1,b,i_2),(b,c,d) \in E \\ 310 & if \quad \exists b,c,e,f \in E_{IDS} : (i_1,b,i_2),(b,c,d),(b,f,e) \in E \wedge c \neq f \\ 100 & else \end{cases}$$

4.2.3 Update

Even though we implement a CRUD interface, we do not offer support for any update operations.

The most important reason is correctness and performance. Updating the source and target of edges has the potential of creating impossible loops, like an edge connecting itself. While this is impossible to do when constructing the edge at first (as it is required that its source and target already exist), this can no longer be guaranteed when the edge is updated. An alternative would be to allow updates, but search for correctness violations (*i.e.*, recursively following the source and target of an edge, we ultimately end up in nodes) after the update was done. This would have a significant, and unpredictable, impact on performance when performing an update for an edge. As an update operation is similar to a subsequent create and delete, which have better complexity, we did not think this is a viable approach. Yet another alternative would be to allow updates again, but only those updates that change the source and target to nodes that existed when the edge was originally created. This prevents correctness violations by construction, though it does not make the operation generally applicable. And since we would need to have a fallback method (*i.e.*, subsequent delete and create) anyway, it might be easier to just always use the fallback method. This also prevents us having to store some kind of causality information, like which elements were created before which other elements.

Another reason is cache management, as also proposed by [20]. If a node can be updated, caches can become invalid, implying some kind of MvS-initiated invalidation protocol for the MvK. While we do not have any significant optimization for this yet, restricting updates has significant potential.

4.2.4 Delete

Finally there are the *delete* operations. The source and target of each edge should always exist in the graph. Therefore, if a deleted node or edge is the source or target of an edge, the edge needs to be recursively removed. The resulting graph should thus be the largest possible subgraph of the original graph, while still being a valid graph. For the delete node operation (D_N), the node itself is removed, and then all connected edges are recursively removed.

$$\begin{gathered}
D_N : \mathcal{G} \times N \to \mathcal{G} \times S \\
D_N(G,i) = (G',s) \\
G = \langle N, E, N_V \rangle \\
G' = \langle N', E', N'_V \rangle \\
N' = N \setminus \{i\} \\
G' \subseteq G \\
\forall G'' \subseteq G : (G' \subseteq G'') \Rightarrow G' = G'' \\
s = \begin{cases} 220 & if \quad i \notin N \\ 100 & else \end{cases}
\end{gathered}$$

The delete edge operation (D_E) operation is similar, but it is guaranteed that no nodes are removed at all. Due to recursive deletions, the resulting set of edges is possibly a subset of the original edges. The resulting graph is again the largest possible (valid) subgraph, with the specified edge removed.

$$\begin{gathered}
D_E : \mathcal{G} \times E_{IDS} \to \mathcal{G} \times S \\
D_E(G,i) = (G',s) \\
G = \langle N, E, N_V \rangle \\
G' = \langle N, E', N'_V \rangle \\
E' \subseteq E \setminus \{(a,i,c) \in E\} \\
G' \subseteq G \\
\forall G'' \subseteq G : (G' \subseteq G'') \Rightarrow G' = G'' \\
s = \begin{cases} 221 & if \quad i \notin E_{IDS} \\ 100 & else \end{cases}
\end{gathered}$$

5 Modelverse Kernel

We will now consider the Modelverse Kernel (MvK), which is responsible for the execution of action code. Execution of action code causes changes to the PTM, which need to be handled by the MvS. As such, the Modelverse Kernel is responsible for the mapping between the user-level and the PTM. Users can create action code constructs directly, thus forming a direct interface to the MvS for the user. Alternatively, users can create models using a formalism which has action code constructs defined (*e.g.*, to define the model semantics).

As everything is modelled explicitly (Axiom IV: Model Everything), both the execution context and instructions to execute are part of the MvS, and can thus be accessed by the MvK and ultimately the user. When executing an action language model, the execution context is modified in the MvS. Therefore, the MvK itself does not have any local state. By making all states and intermediate steps explicit, we obtain enhanced debugability and introspection. This furthermore contributes to Axiom I: Forever Running, as it allows action code to modify other action code (*i.e.*, self-modifiability).

We first introduce the notion of transformations for our graph, subsequently called graph transformations [1]. Such transformations consist of a matching part, which we will use to determine if the execution context is well-defined, and a rewriting part, which we can use to define the action language semantics by defining transformations of the execution context.

5.1 Graph transformations

Before we can use graph transformations in our well-formedness check and semantical definition, we need to define them first. We need to bridge the gap between graph transformations and the CRUD operations defined by our MvS interface.

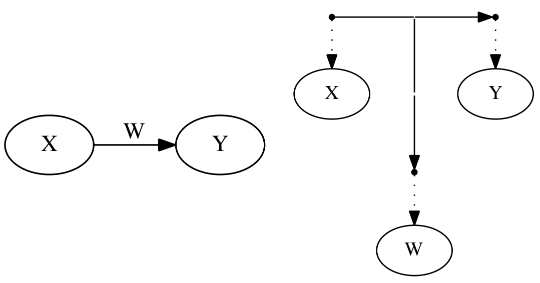
For each transformation rule, it is possible to decompose it in four distinct (sequentially ordered) components. The first two are read operations, which are used for the matching, and the last two are create and delete operations, which are used for the rewriting.

1. **Positive read** operations are used for elements which are present before and after execution of the transformation rule. They are used for finding a possible match during the matching phase. Note that all elements need to be matched, even those that are about to be removed. All elements that are now matched, can be used during the rewriting phase. Elements that are simply required for the match, but without any changes to them, are visualized by black, solid lines.

2. **Negative read** operations are used for the negative application conditions. Such elements should not be present before application of the transformation rule. If they are present in a match, the match is considered incorrect and another match is searched for. Elements which are searched for here, can of course not be used during the rewriting phase, as we explicitly required that they are absent. They are visualized by a red, dotted line.

3. **Delete** operations are used for elements that need to be removed during the rewriting phase. Elements which should be removed, should also be matched in the positive read operation. These elements are visualized by a blue, dashed line.

[1]They are called graph transformations, though are different from the usual meaning of graph transformations in the literature. While the idea is similar, we provide a different mapping as we do not work on Typed Attributed Graphs (TAGs).



(a) Shorthand notation (b) Expanded notation

Figure 5.1: Shorthand notation and equivalent expanded notation.

4. **Create** operations are used for elements that need to be created during the rewriting phase. Because the element is newly created, it does not need to be matched by a positive read operation. However, we do not require them to be absent either. They are visualized by a green, wide solid line.

Each rule can be written in the following form, assuming success status, thus mapping to our previously defined formalization of the MvS. If an error is encountered, it is propagated to the user.

$$\frac{\begin{array}{c} PositiveRead_A(G) \\ NegativeRead_A(G) \\ G' = Create_A(G) \\ G'' = Delete_A(G') \end{array}}{G \xrightarrow{step_A} G''}$$

Each rule uses the matched elements, which get bound during application. As such, the *PositiveRead* operation binds the variables, which are then used in the *NegativeRead* to detect invalid matches, in the *Delete* to delete elements, and in the *Create* to create new elements.

For conciseness, we define the shorthand notation for graph elements shown in Fig. 5.1a, equivalent to Fig. 5.1b, meaning:

$$\begin{gathered} (X_{node}, W_{link}, Y_{node}) \in E \\ (W_{link}, e, W_{node}) \in E \\ N_V(X_{node}) = X \\ N_V(Y_{node}) = Y \\ N_V(W_{node}) = W \end{gathered}$$

If X, Y, or W is not shown in the shorthand notation, then the N_V mapping is unconstrained, and might not even exist.

An example of the mapping between the shorthand notation and the previously defined semantics is given next. We explain the transformation shown in Figure 5.2. First, the *success* status code is stored in s (equation 5.1), to shorten subsequent rules. All parts of the rule are assumed to result in a success status code. The positive read operations start at equation 5.2, followed by the negative read operation at equation 5.6. Now that all nodes and edges are bound, the create operation creates the necessary links starting from equation 5.7. From equation 5.10 to the end, operations try to match the edge to delete at a finer granularity and delete it in equation 5.18.

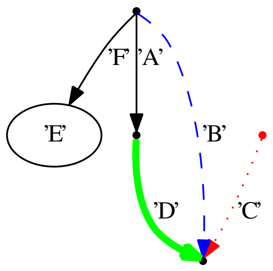


Figure 5.2: Example graph transformation which is expanded

$$s = (100, \_) \tag{5.1}$$
$$R_{dict}(G, a, "A") = (b, s) \tag{5.2}$$
$$R_{dict}(G, a, "B") = (c, s) \tag{5.3}$$
$$R_{dict}(G, a, "F") = (e, s) \tag{5.4}$$
$$R_V(G, e) = ("E", s) \tag{5.5}$$
$$\nexists d : R_{dict}(G, d, "C") = (c, s) \tag{5.6}$$
$$C_{NV}(G, "D") = (G', f, s) \tag{5.7}$$
$$C_E(G', b, c) = (G'', g, s) \tag{5.8}$$
$$C_E(G'', g, f) = (G''', h, s) \tag{5.9}$$
$$\langle V_1, s\rangle = R_O(G''', a, s) \tag{5.10}$$
$$\langle V_2, s\rangle = R_I(G''', c, s) \tag{5.11}$$
$$i \in V_1 \tag{5.12}$$
$$i \in V_2 \tag{5.13}$$
$$\langle V_3, s\rangle = R_O(G''', i, s) \tag{5.14}$$
$$j \in V_3 \tag{5.15}$$
$$R_E(G''', j) = ((i, k), s) \tag{5.16}$$
$$R_V(G''', k) = ("B", s) \tag{5.17}$$
$$D_E(G''', i) = (G'''', s) \tag{5.18}$$

$$\overline{\qquad G \xrightarrow{step_A} G'''' \qquad}$$

Note that this does not explicitly remove all parts of the edge. Specifically, the node containing the data value still remains. This is because it might still be referenced from somewhere else, and deleting that might have serious repercussions. As a safety measure, only the link itself is removed. All elements that are no longer reachable from the root will later be removed in the garbage collection phase.

The current notion of graph transformations should not be confused with the notion of model transformations, which the user can use. The graph transformations we have defined here, are merely a conceptual construct, used to formalize the semantics of action language constructs. It is therefore not mandatory for an MvK implementation to implement the semantics as if it were a graph transformation. On the other hand, model transformations, which are implemented on top of action language constructs, will be usable by the user, and as such are really implemented as transformations. Furthermore, model transformations will be at a level closer to the user (*i.e.*, not on raw graphs), and will therefore be typed. We will not discuss model transformations any further in this technical report, as this is part of future work.

5.1.1 Performance

It is important to mention that our graph transformations do not use the notion of types. As we are working on simple graphs, which do not have a real notion of type, and it can therefore not be used. This implies that *nodes* in the transformation rules can as well be edges, since the semantics of a point in the tranformation rules is simply an identifier from *IDS*. Conversely, this might imply a performance impact, as the only basis to determine a match is the use of primitive values, and edges between specific nodes. While this is a concern relating to Axiom II: Scalability, it is not a fundamental problem for the following reasons:

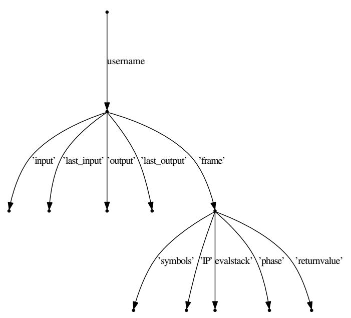


Figure 5.3: Graph to match as execution context. Some nodes might be identical.

1. We start from a pivot, which is the Modelverse Root previously defined. All MvK instances will know which node to use for this, and therefore there is already a place where the matching can start.
2. At most one match exists. This means that we can already stop searching as soon as a single match is found.
3. All edges have a constant on them, which needs to be unique. Therefore, no trackbacking is required as soon as the correct edge is found: each edge will be identifiable.
4. A primitive operation exists to read out the aforementioned edges: the R_{dict} operations. If this operation is implemented in $O(1)$ (*e.g.*, using hashmaps), this means that the complexity of finding a match is unrelated to the size of the host graph.

Combining all of this information, we can write a simple algorithm for each rule, which starts from the Modelverse Root, and just performs a serie of R_{dict} operations. As there is only one possible result for that operation, we do not need to rely on backtracking.

Some exceptions exist to these findings though, such as the accessing of variables in the symbol table. These do not use values on the edge that are in $\mathbb{U}$, and therefore require more advanced algorithms.

5.2 Execution context

We specify the structure of the execution context by defining a graph that has to be matched. If the graph is matched, the execution context is valid and execution is possible if the current instruction is valid. If no match can be found for the specified user, the user's execution context is invalid and execution is impossible. If multiple matches are found, the execution context is also invalid, and results will be undefined. We make no distinction between *no execution context* (*i.e.*, nothing at all) and *corrupt execution context* (*e.g.*, a single missing link). In either case, no execution is possible. During normal operation, the user is unable to corrupt or remove its own execution context, as all action language primitives are guaranteed to keep the execution context in a valid state. But in case introspection or self-modifiability is used (*i.e.*, if an intentional change is made by the user), it is possible to alter, and possibly corrupt, the execution context. This is possible because the execution context is itself again another model in the MvS, and it can be manipulated like any other. We do this to enable self-modifiability, introspection, reflection, and debugability, which can now be performed directly on the graph. For debugging it is even possible for another (privileged) user to debug the state of another user, or process.

A valid execution context is one that is matched by the structure in Fig. 5.3. At the top of the structure sits the Modelverse root node, which is a node that is known to the MvK. From this root node, there is a link to all user root nodes, containing the name of the user. In our figure, *username* has to be interpreted as a variable for the transformation. From the user root, there are links for *input* and *output* lists, and a *"frame"* link. These input and output links come with both an initial link, and the *last_* variant, which points to the last element of the list. The last element will always be empty (have no value), but needs to be there to guard for the case of an empty list. Each element in such a list will have a *"value"* link, which points to the actual value, and a *"next"* link, which points to the next element in the list. The exception to this is the element pointed to by the *"last_"* element, which is the empty placeholder.

The destination of the *"frame"* link is the currently active execution frame for that user. Each execution frame has several outgoing links.

First is the *"symbols"* link, which points to the symbol table. A symbol table is a node which is interpreted as a dictionary, where all outgoing edges have a unique key. The symbol table can then be accessed using the R_{dict} CRUD operation of the MvS. In this case, the key is the variable definition in the code being executed. The destination of the edge is the current value of the specified variable. It is not possible to save this variable in the executing code itself, as the code can possibly be executed by different users simultaneously (Axiom X: Multi-User).

Second is the *"IP"* link, which points to the current action code primitive being executed. It is similar to an Instruction Pointer, with the exception that it does not advance linearly, nor is there a default direction. Every instruction primitive is responsible to update the instruction pointer.

Third is the *"evalstack"* link, pointing to the evaluation stack. In this stack, instructions are stored, which need to be made in the same scope. Such a structure is necessary because we do not use compiled bytecodes which modify a stack. For example, for the execution of an *If* construct, we first need to evaluate the condition. In such "stack-based" languages, the result of the condition is first put on the stack by the appropriate bytecodes. Only then a bytecode concerning the *If* construct is encountered, which consumes the evaluated value on the stack. In our language, the *If* is encountered first, which then explicitly states to evaluate the condition first (by moving the "IP" link), and come back as soon as it is evaluated (by putting it on the evaluation stack).

There is also the *"phase"* link, allowing for a distinction between the different sub-states in the evaluation of a primitive. For example in the *If* construct, a distinction between the *"evaluate condition"* and *"branch on value"* phases is necessary. To make this possible, the phase keeps the current state of the evaluation of that specific execution primitive.

Finally, there is the *"returnvalue"* link, which links to a node which contains the value from the previous execution. Each instruction primitive can read and update this link. It is used for the exchange of temporary values between different instructions. In contrast to languages which use a stack, there is only one temporary variable in our language. This offers us a slightly more efficient implementation of most constructs, due to avoiding the use of a list. Some constructs get more complex (though not necessarily slower, performance-wise), such as those where it is natural to evaluate multiple values sequentially.

Some additional links might be present on the frame, and their use is mandatory, though they are not required for a well-defined execution context. These links are the *"prev"* and *"variable"* links. The "prev" links to the previous execution frame, that is, the invoker of the function for which the frame is created. The "variable" link is used during assignment, as we need two evaluated elements at the same time: the variable to write to, and the value to assign. If these links are not present at the time where they are necessary, the execution context is considered to be corrupt. These optional links could be made mandatory, by setting making them point to an empty node if they are not necessary.

The execution context is well-defined if exactly one such match is found for a given user. No matches means that there is no execution context with all required elements (*i.e.*, either corrupt or completely missing). Multiple matches indicate non-determinism and are therefore not allowed. Additional elements, though not indicated here, are allowed, as they do not interfere with the match. These elements should be considered implementation-dependent and should not be used for the implementation of functional requirements.

Apart from the user-specific execution context, there is also a global symbol table, stored as if it were the *"__global"* user. This symbol table is shared by all users, and is accessed if a variable could not be found in the local symbol table of the current execution frame.

5.3 Execution primitives

What remains is the semantics of each of the action language constructs. For each construct, defined in $\mathbb{A}$, the required modifications on the execution context needs to be defined.

As proposed in previous sections, graph transformations are used to define the semantics. 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 (Axiom X: Multi-User), interleaving is necessary between them to guarantee fairness. This also prevents uninterruptible loops (Axiom I: Forever Running), as another (privileged) user can then always halt the execution of another user. For performance reasons (Axiom II: Scalability), an MvK can ignore updates to the execution context (*e.g.*, by not propagating them to the MvS, or by implementing compiled operations). But this comes at the cost of debugability, introspection, and self-modifiability. Hybrid approaches are supported, meaning that some functions will be called without modifications to the execution context (*e.g.*, primitive operations such as integer addition), whereas others modify to the execution context.

A step function is defined for each user, which applies the only applicable rule.

$$\frac{\begin{array}{c} G \xrightarrow{step_A} G' \vee \\ G \xrightarrow{step_B} G' \vee \\ \ldots \end{array}}{G \xrightarrow{step} G'}$$

The interleaving of different users, and thus of different steps, is not specified, as long as there is some fairness between all users. This allows for the definition of primitive operations in the Modelverse Kernel, which consist of several (atomically executed)

Construct	Name	Mandatory	Executable	Meaning (informal)
If	cond	Yes	Yes	Condition
	true	Yes	Yes	Block to execute if condition is True
	false	No	Yes	Block to execute if condition is False
	next	No	Yes	Next instruction after True/False block
While	cond	Yes	Yes	Condition
	body	Yes	Yes	Body to execute while condition is True
	next	No	Yes	Next instruction after condition is False
Break	while	Yes	Yes	While construct that should be broken
Continue	while	Yes	Yes	While construct that should be continued
Access	var	Yes	Yes	Variable to access
Resolve	var	Yes	No	Variable definition to access
Assign	var	Yes	Yes	Variable to assign to
	value	Yes	Yes	Value to assign to variable
	next	No	Yes	Next instruction after assignment
Call	func	Yes	Yes	Function signature to call
	next	No	Yes	Next instruction after function call returned
	params	Yes	No	First parameter, linking to a *Parameter*
	last_param	Yes	No	Last parameter, linking to a *Parameter*
Parameter	name	Yes	No	Name of the parameter, used to link with the formal parameters
	value	Yes	Yes	Instructions to evaluate as parameter
	next_param	No	No	Next parameter to be evaluated (optional if this is the last parameter)
Return	value	No	Yes	Value to return
Const	node	Yes	No	Node containing the constant to access
Input		N/A	N/A	N/A
Output	value	Yes	Yes	The node to output

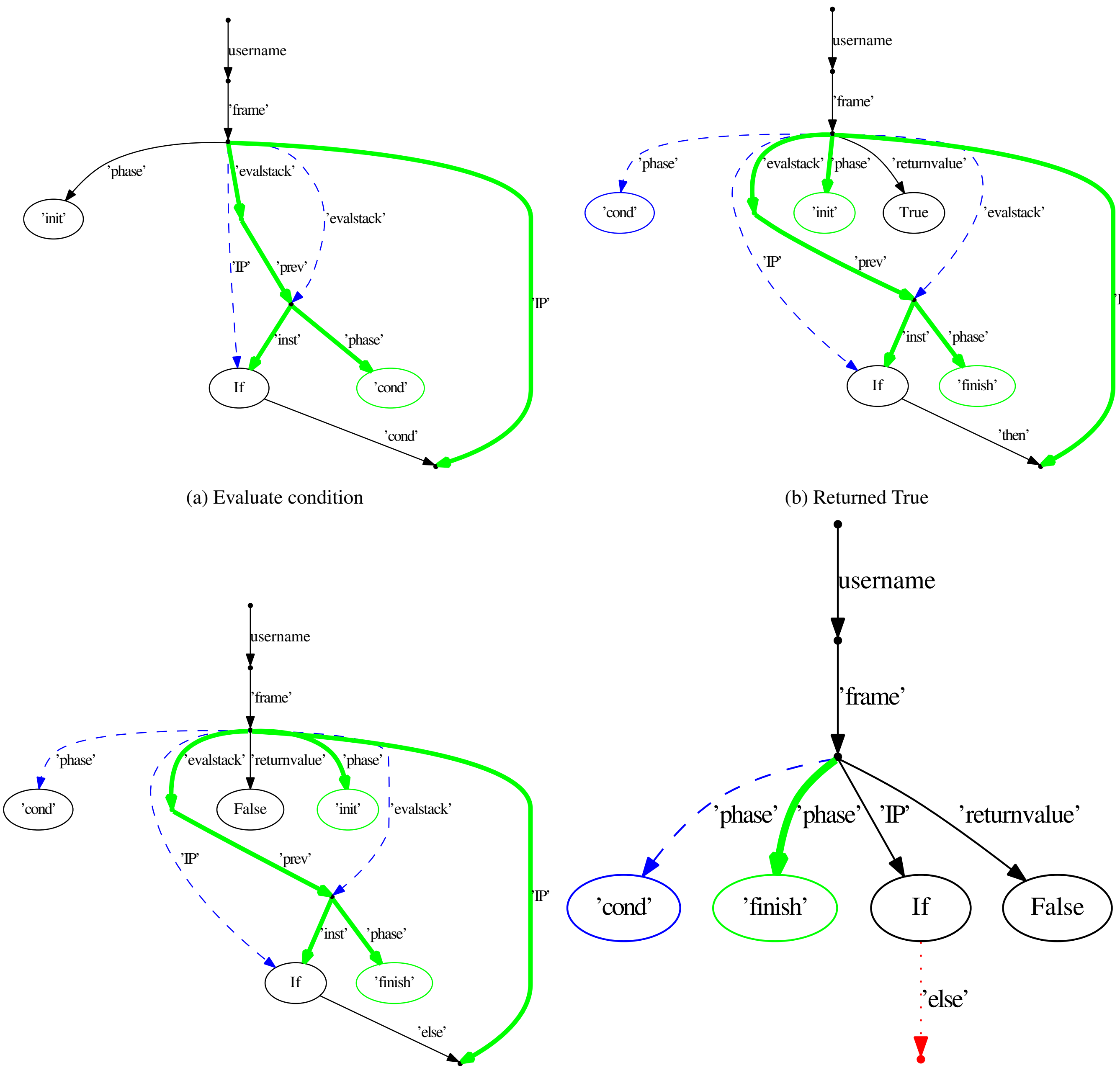
Table 5.1: Outgoing link specification.

instructions. Such primitives can then be used for performance reasons (Axiom II: Scalability), but also as a core function (Axiom IV: Model Everything).

In Table 5.1, we present an overview of all specified outgoing links for each primitive element. A construct is valid if all mandatory elements are present. Links which guide the instruction pointer, require the target of the link to be executable (*i.e.*, be another primitive construct, $\in \mathbb{A}$). If that is not the case, execution will terminate.

Normally, the action language constructs are created by a different tool, such as a HUTN MvI, which will guarantee that the constructs are well formed. But it is possible for users to access all parts of the MvS, thanks to Axiom IV: Model Everything, and therefore to manually create (or alter) action language constructs. Such actions cannot automatically be checked for correctness, due to our lack of typing: there is no metamodel for the primitive action language constructs. And since there is no metamodel, there is no constraint on the graph. Although counter-intuitive, this is actually what we want: unconstrained modifications on the raw model representation, thus allowing model management operations. In the next chapter, we will add a layer on top of all this, which is typed and more constrained. Notwithstanding, it is possible to create a function which manually checks whether or not a construct is well-defined, using the information from Table 5.1.

5.3.1 If



(a) Evaluate condition

(b) Returned True

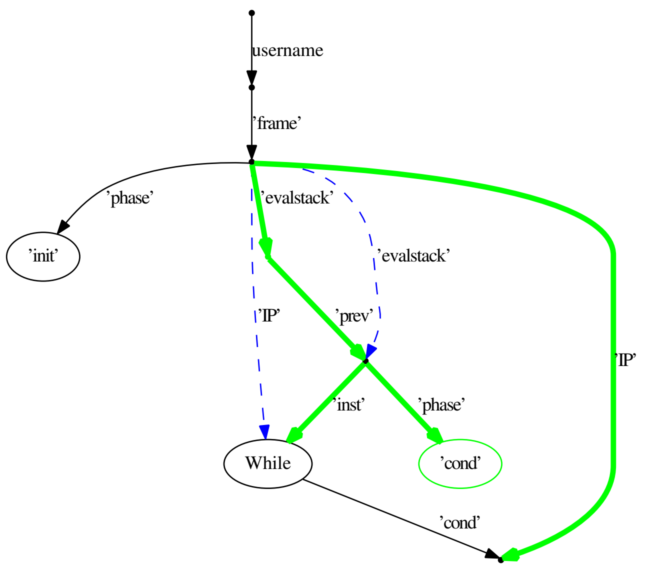
(c) Returned False and there is an 'else' block

(d) Returned False but there is no 'else' block

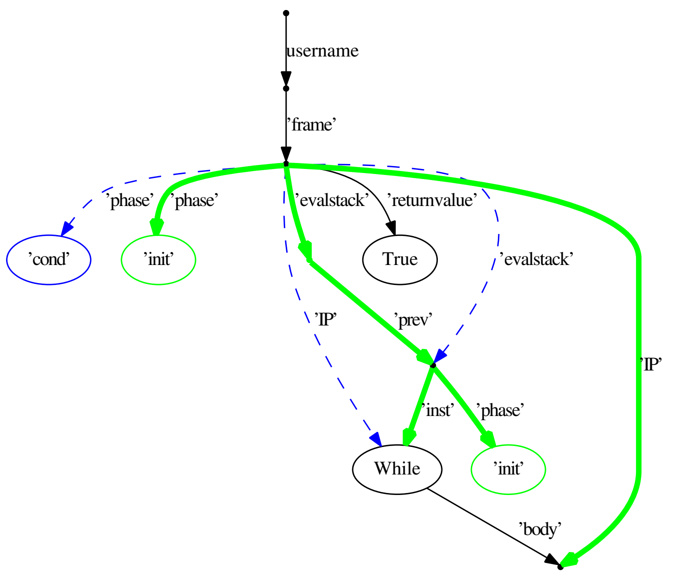
Figure 5.4: If branch rules

The *If* construct will first evaluate the condition (*cond* link) by moving the instruction pointer there. It signals that it should be executed again afterwards, but now in phase *cond*, by putting this on the evaluation stack (Figure 5.4a). As soon as the condition is evaluated, and the *If* popped back from the stack, the return value (of the condition) can either be True or False. If it is True (Figure 5.4b), the *then* link is executed, and the *if* is pushed on the stack again, but now in the final phase *finish*. This is the phase which signals to another rule that this operation has finished, and the next instruction can be loaded. If it is False, and there is an *else* link (Figure 5.4c), it is executed, similar to the previous case. If it is False, but there is no *else* link (Figure 5.4d), the *If* is marked as completed immediately, without any subsequent actions.

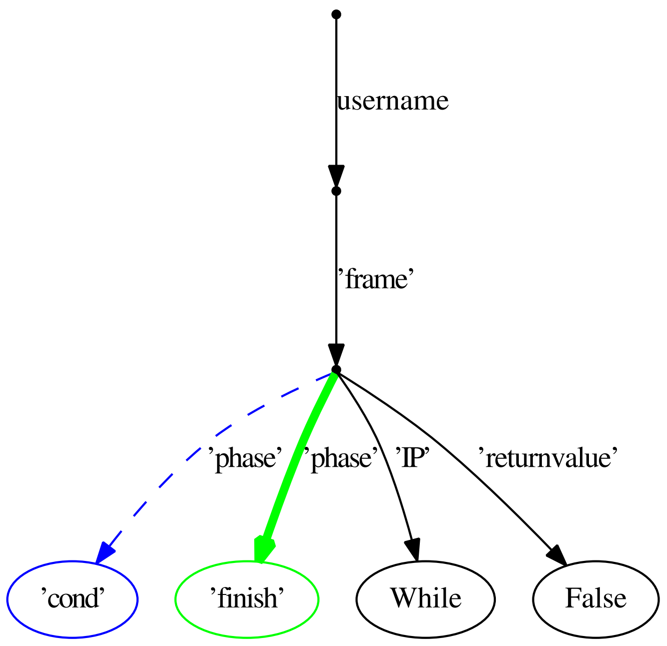
5.3.2 While



(a) Evaluate the condition of the While



(b) Condition was true



(c) Condition was false

Figure 5.5: While loop rules

The *While* construct will first evaluate the condition (*cond* link) by moving the instruction pointer there. It signals that it should be executed again afterwards, but now in phase *cond*, by putting this on the stack (Figure 5.5a). As soon as the condition is evaluated, and the *While* popped from the stack, the return value (of the condition) can either be True or False. If it is True (Figure 5.5b), the *body* link is executed, and the *While* is pushed on the stack again, but with its phase set to *init*. This way, the while construct will again be executed after the body has terminated. By setting the phase to *init*, we effectively cause looping, as the condition will again be evaluated, and, depending on the result, the body gets executed once more. If it is False (Figure 5.5c), the *While* is immediately marked as finished and the body is not executed.

5.3.3 Break

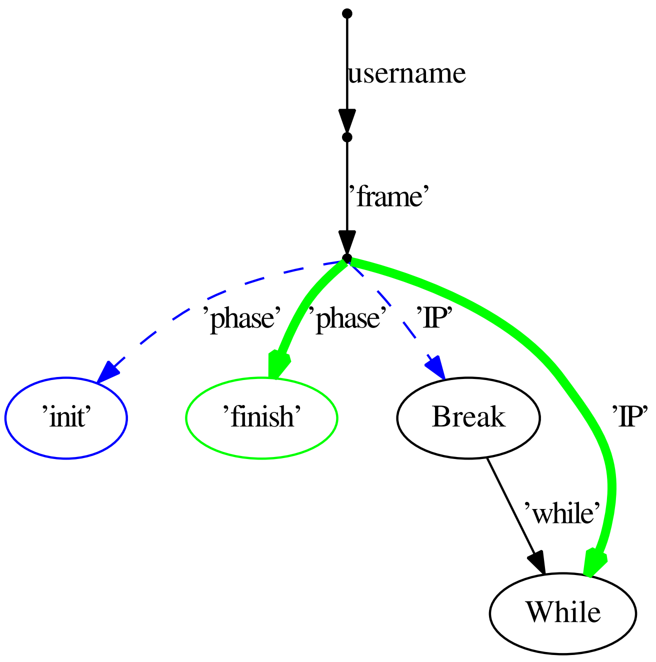


Figure 5.6: Break rule

The *Break* construct will move the instruction pointer back to the *While* construct it belongs to (Figure 5.6). The phase is set to *finish* to indicate that the loop has finished. This prevents the condition evaluation and marks the end of the while loop.

5.3.4 Continue

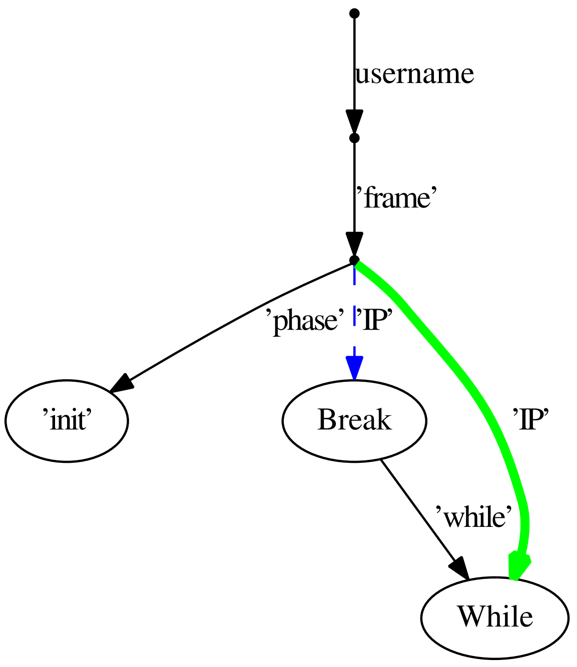
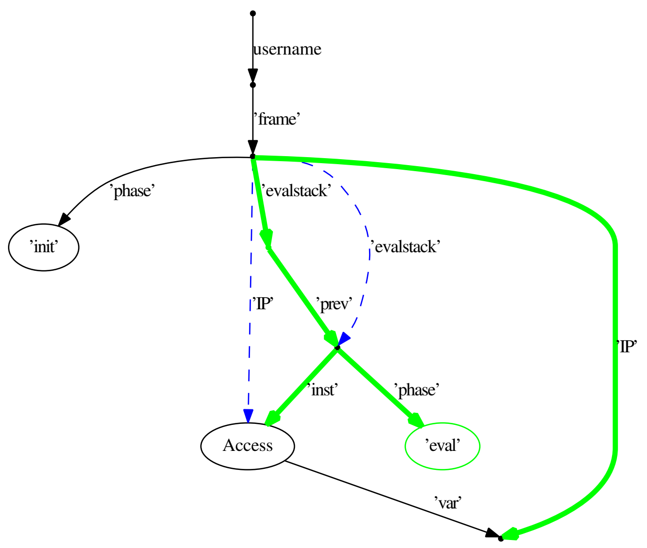


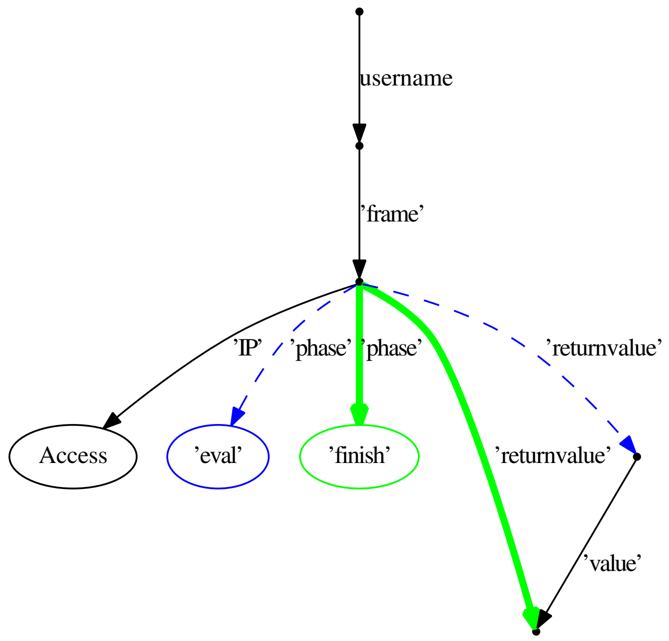
Figure 5.7: Continue rule

The *Continue* construct will move the instruction pointer back to the *While* construct to which it belongs (Figure 5.7). The phase is set to *init* to indicate that the loop needs to continue. This causes the condition to be evaluated again, indicating the next iteration of the loop.

5.3.5 Access



(a) Evaluate variable to access

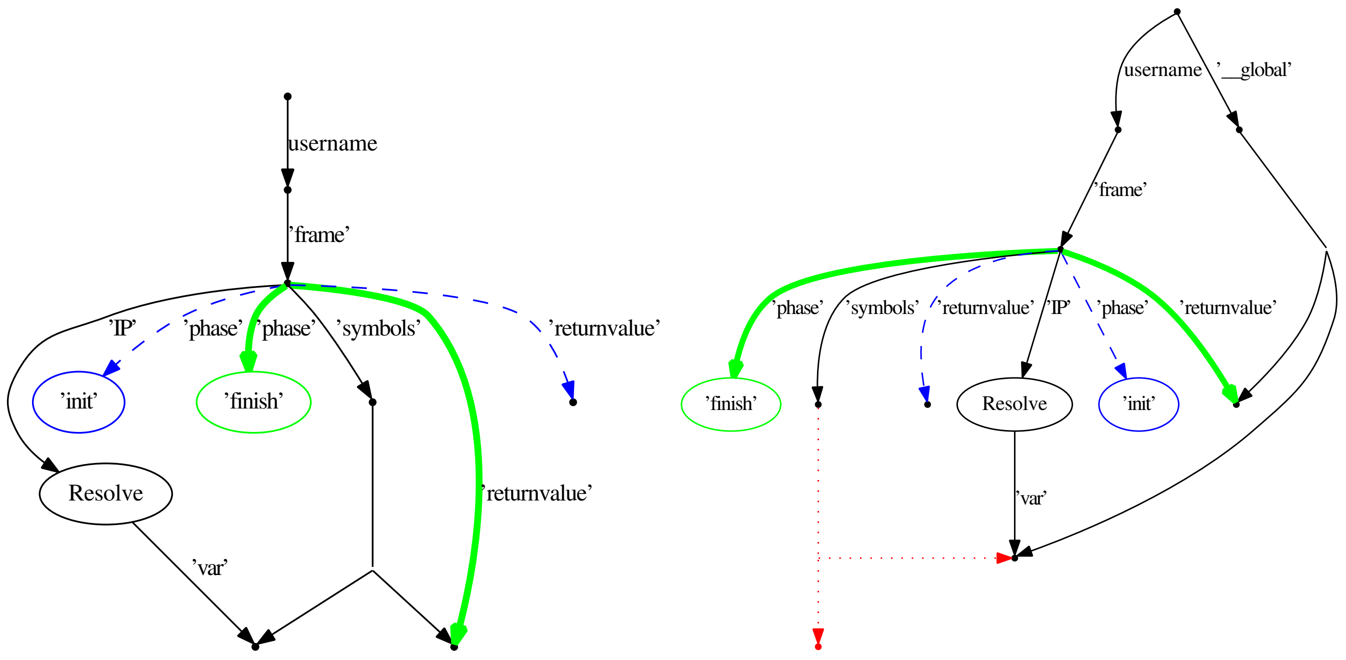


(b) Access evaluated variable

Figure 5.8: Variable dereference rules

The *Access* construct will move the instruction pointer to the variable which has to be resolved first (Figure 5.8a). It signals that it needs to be executed again after the variable was resolved, by putting itself on the evaluation stack. After resolution of the variable, the value of the variable is accessed and set as the new return value (Figure 5.8b).

5.3.6 Resolve



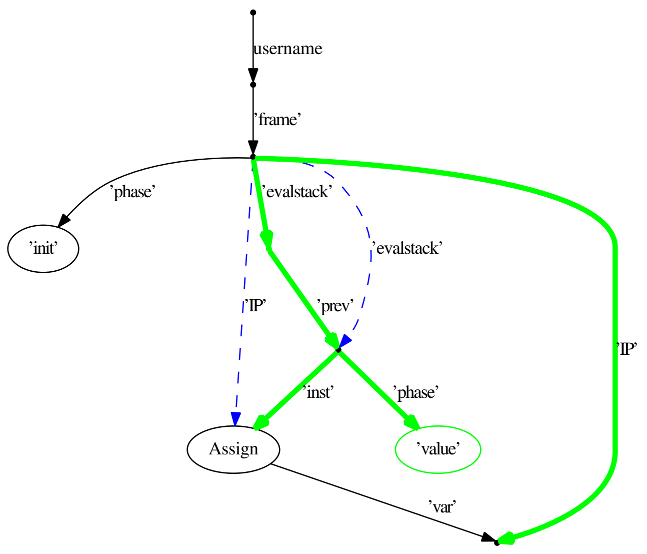
(a) Access the variable from the local symbol table

(b) Access the variable from the global symbol table

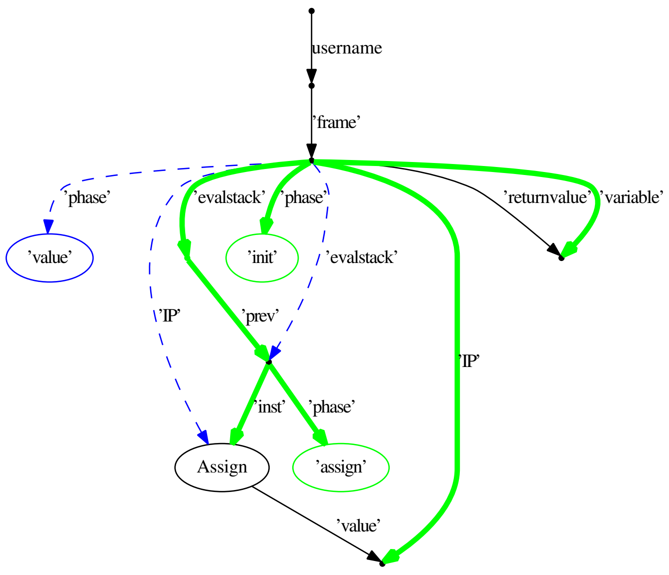
Figure 5.9: Resolution rules

With the *resolve* rule, a variable is looked up in either the local (Figure 5.9a) or global (Figure 5.9b) symbol table. The variable in the symbol table will be set as the returnvalue. The local symbol table has priority over the global symbol table. Note that the returned value is only a reference, similar to the lvalue in parsers. A further *Access* is required to read out the actual value.

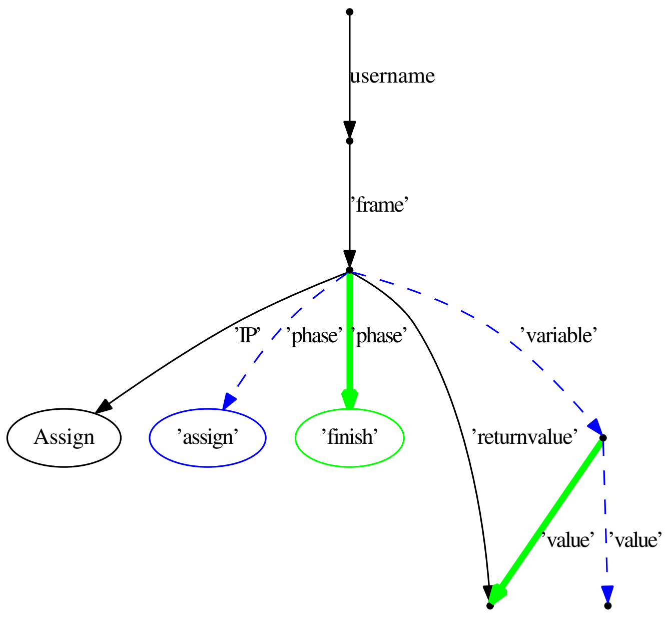
5.3.7 Assign



(a) Resolve the variable to assign to



(b) Evaluate the value to assign

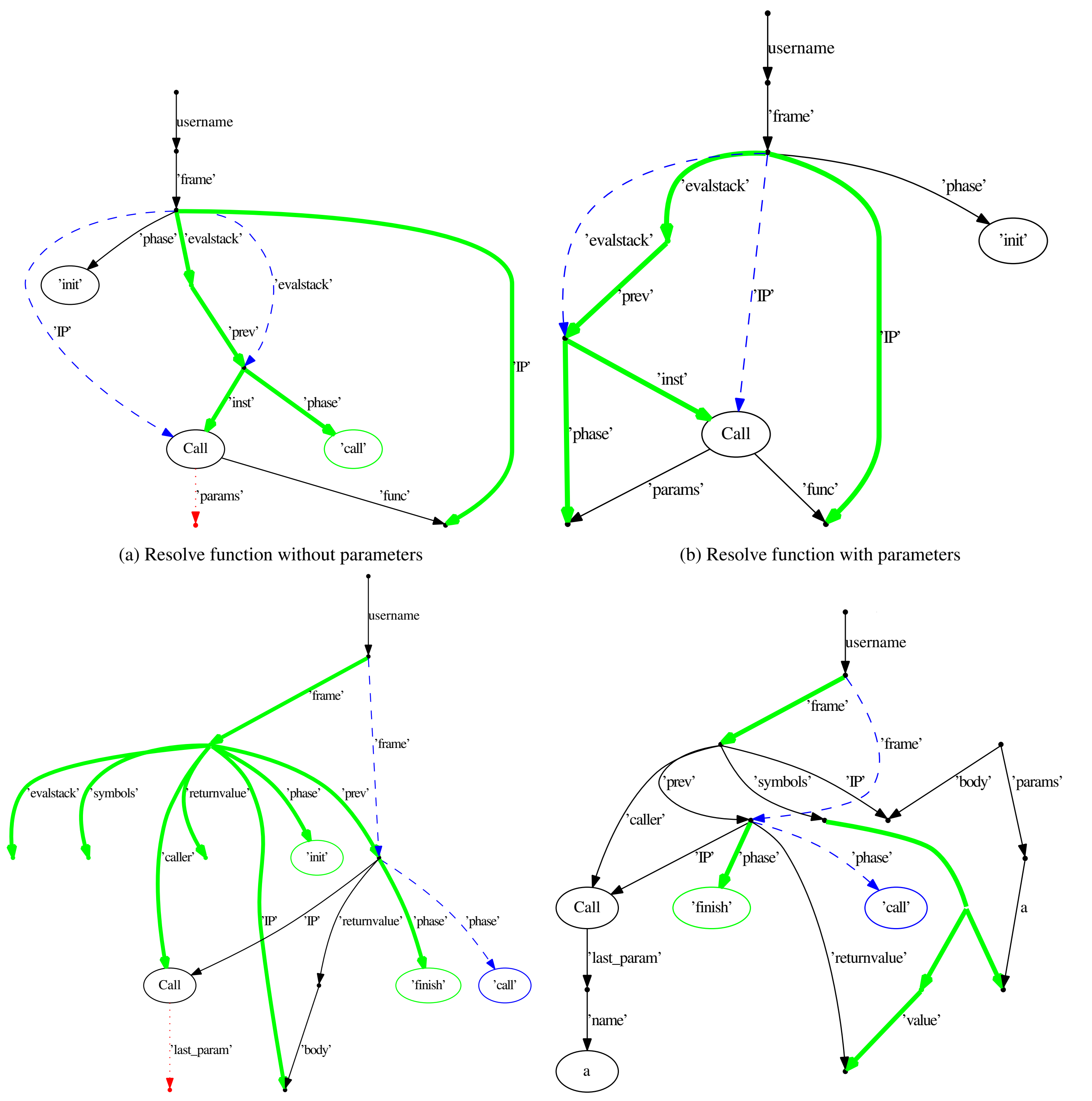


(c) Assign the value to the variable

Figure 5.10: Assignment rules

The *Assign* rule will first evaluate the variable (Figure 5.10a), as it will first need to be resolved. After resolution (Figure 5.10b), the found value is stored in a temporary link from the frame (*variable* link). The instruction pointer is moved to the value that will be assigned, as it will also need to be evaluated. After the value is evaluated (Figure 5.10c), the value link in the stored variable is changed to the evaluated value.

5.3.8 Function call



(a) Resolve function without parameters

(b) Resolve function with parameters

(c) Execute call with no parameters

(d) Execute call with parameters

Figure 5.11: Function call rules for resolution and execution

A *Call* construct has different paths, depending on how many parameters there are. The distinct situations are:

1. **No parameters**: in this simple case, the method is first resolved by moving the instruction pointer there, and the call is already put on the stack (Figure 5.11a). After the function is resolved (Figure 5.11c), the call is made by creating a new execution frame and making it the active frame.

2. **One parameter**: similar to the previous situation, the function is first resolved (Figure 5.11b), but instead of putting the *call* on the stack, the first parameter is used. Afterwards (Figure 5.12b), the stack is created for the resolved function,

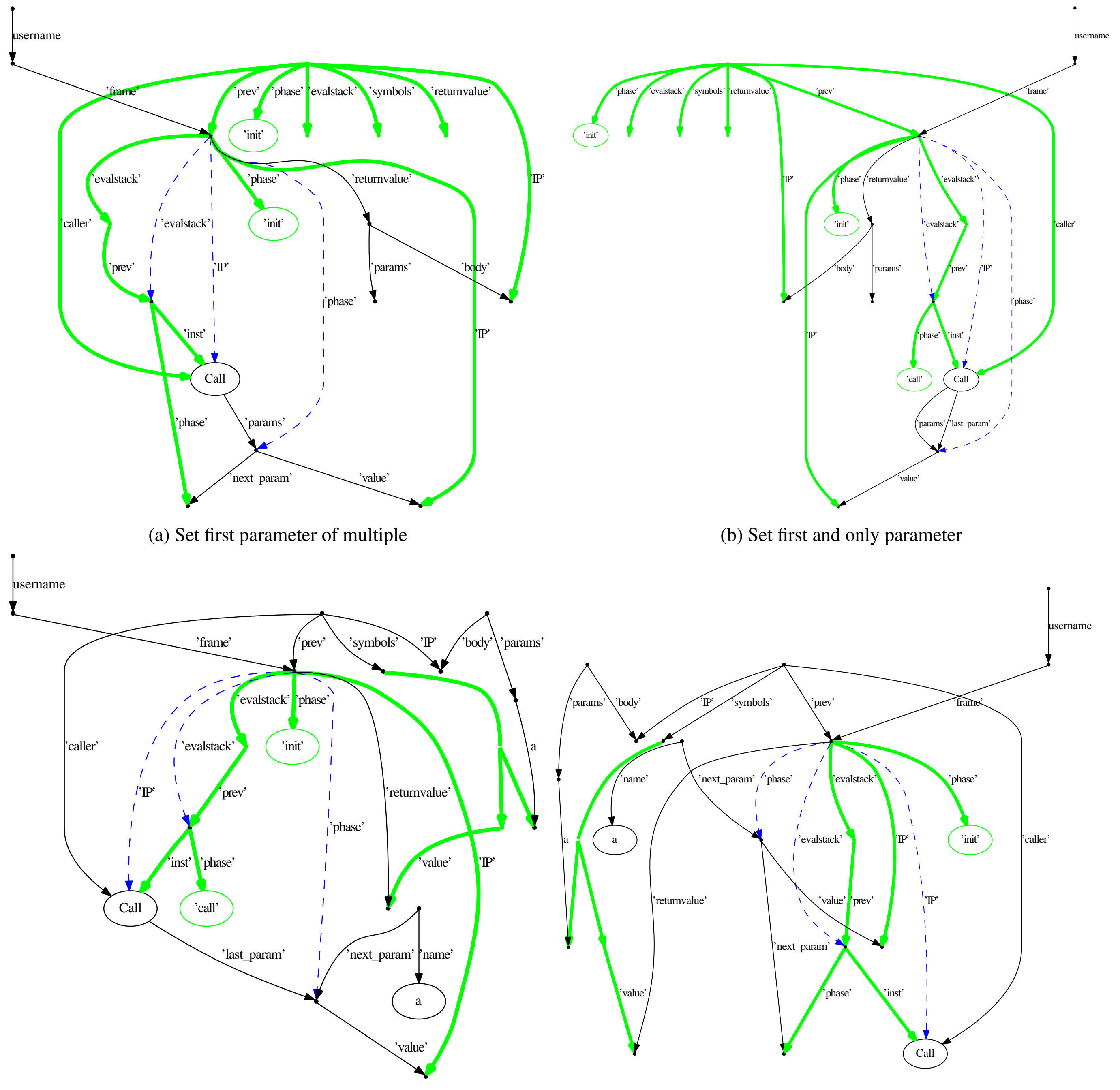
the instruction pointer is set to evaluate the argument, and the *call* is put on the stack. When the parameter is evaluated (Figure 5.11d), the result is put in the symbol table of the new execution frame and the new frame is made active.

3. **Two parameters**: similar to a single parameter, the first parameter is again put on the stack for after the function resolution (Figure 5.11c). When evaluating the first parameter (Figure 5.12a), the *next_param* parameter is put on the stack, instead of the *call* phase. The second parameter is already the last parameter, so we then put the *call* on the stack (Figure 5.12c). Finally, the function is called as with only a single parameter (Figure 5.11d).

4. **More than two parameters**: similar to two parameters, but with an iteration rule (Figure 5.12d) for all parameters except the first and last. This iteration rule simply evaluates the parameters in order of their *next_param* links.

In all cases, the *finish* is put on the stack during the call to the function. As soon as the called function has finished, it will invoke a return and thus pop the active execution frame. This will make the current frame active again, which will then progress towards the next instruction.

Parameter passing happens through the use of both named variables and positional parameters. However, the positional parameters are only used to determine the evaluation order, and not for binding of actual to formal parameter. It is possible for a front-end to offer positional parameters, by automatically mapping them onto their formal parameters.



(a) Set first parameter of multiple

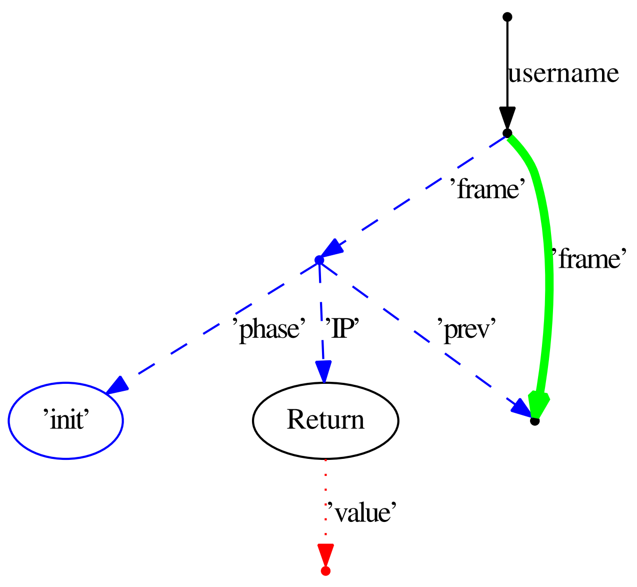
(b) Set first and only parameter

(c) Set last parameter of multiple

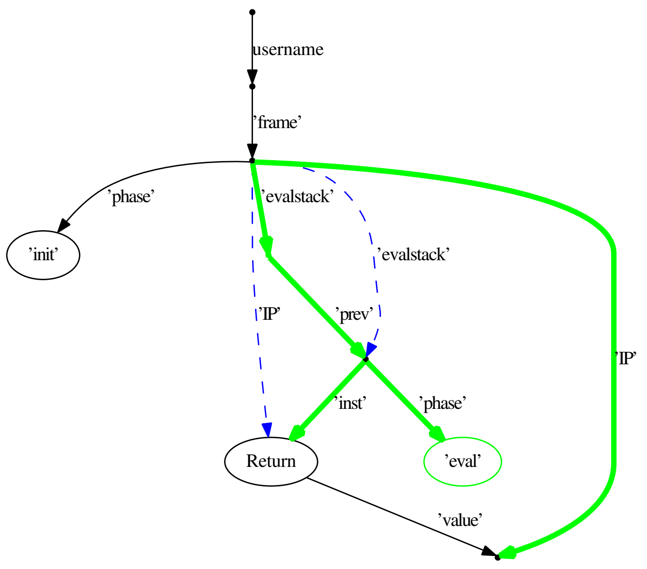
(d) Set next parameter

Figure 5.12: Function call rules for parameter evaluation

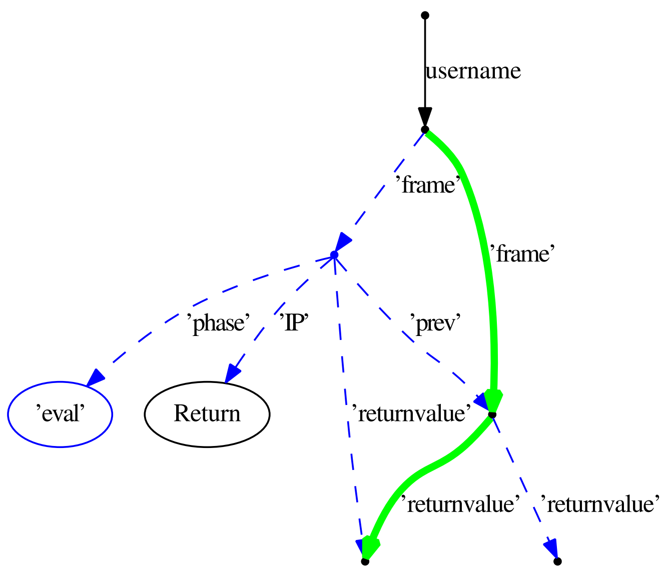
5.3.9 Return



(a) Return without value



(b) Evaluate the value of the return

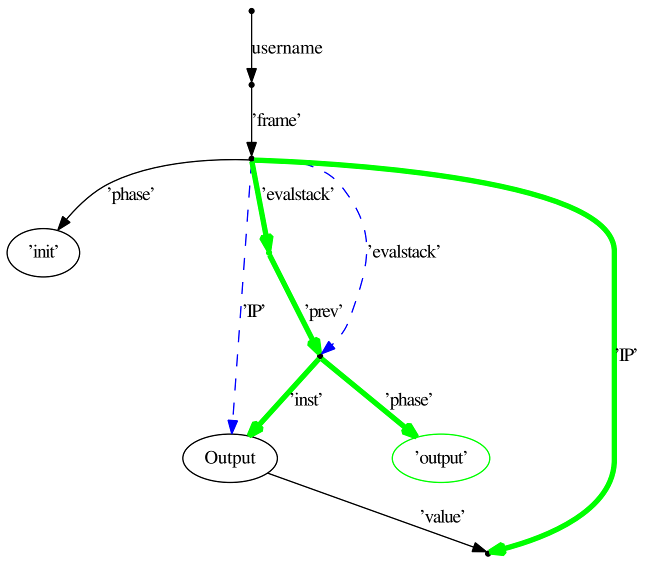


(c) Return with the evaluated value

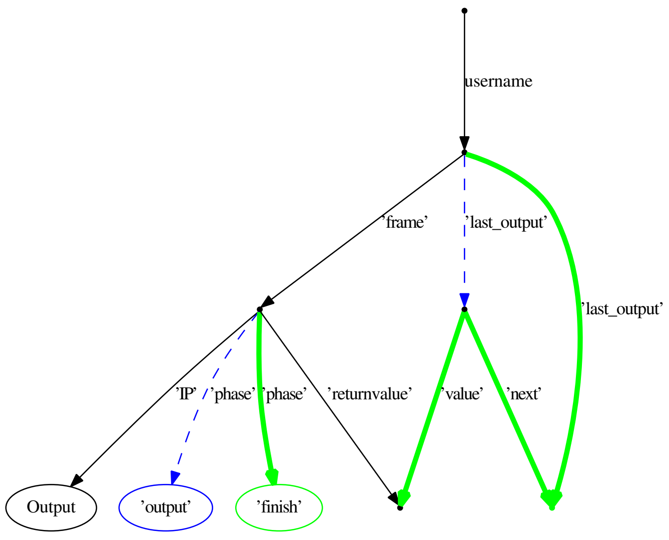
Figure 5.13: Return rules

For the *Return* construct, there are again two options: either there is a value to return, or there is none. If there is no return value (Figure 5.13a), the current execution frame is removed and the previous one is made active again, without touching the return value of the underlying frame. If there is a return value (Figure 5.13b), it is first evaluated by moving the instruction pointer there. After evaluation (Figure 5.13c), the evaluated value is stored in the returnvalue of the previous frame, and the current frame is deleted.

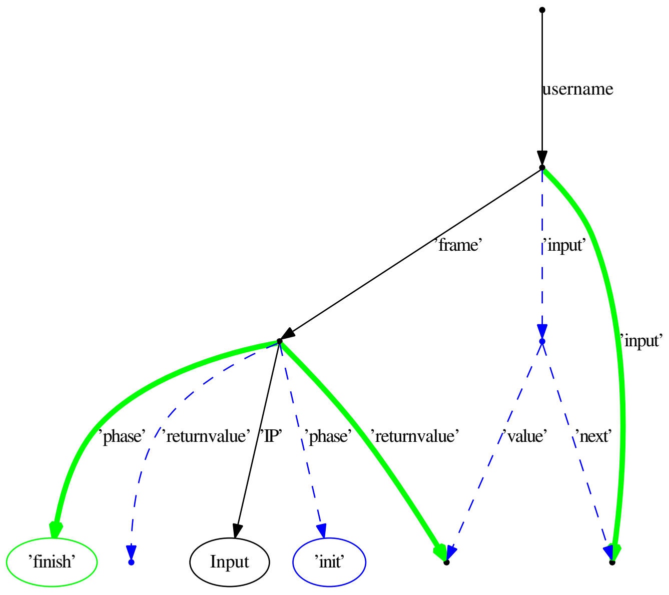
5.3.10 Input and Output



(a) Output rule evaluates value



(b) Output rule outputs value



(c) Input rule consumes input

The *Output* construct will first evaluate the element the 'value' link points to (Figure 5.14a), and afterwards it puts the returnvalue in the output queue (Figure 5.14b).

The *Input* construct will read the value that is in the input queue and put it in place of the returnvalue. No evaluation whatsoever is done on the values.

5.3.11 Constant access

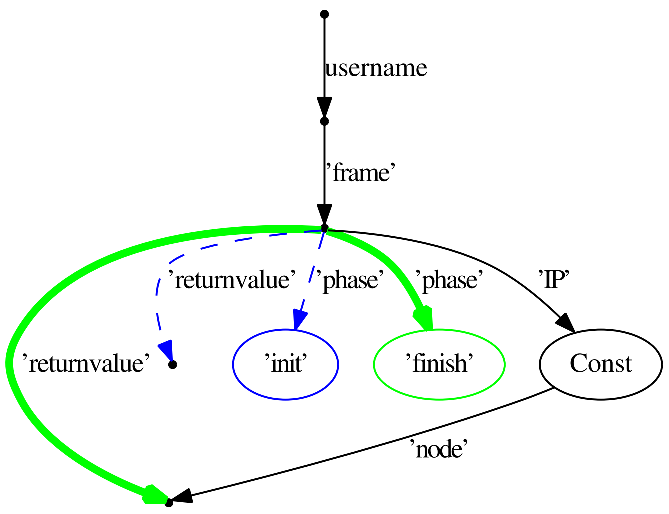
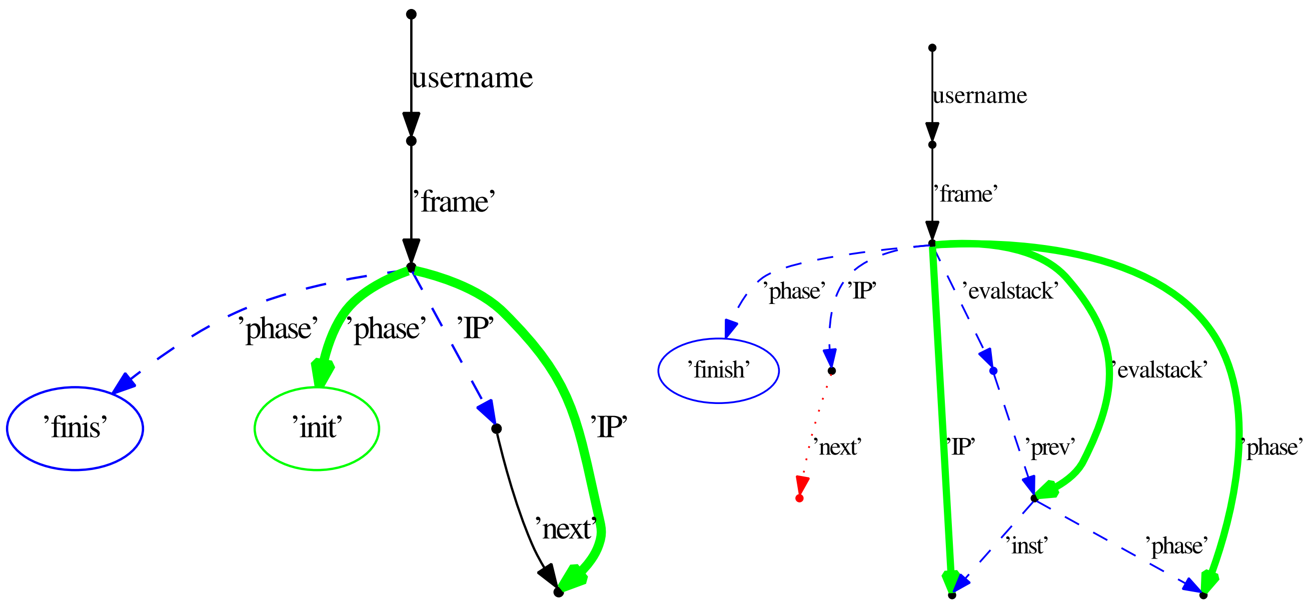


Figure 5.15: Constant access rule

The *Const* construct is used for constants, which are closely linked to the primitive data types presented in the Modelverse State. It is only used as an 'executable wrapper' for a literal: evaluation of this construct will yield the contained node (Figure 5.15). The phase is also set to *finish*, to indicate termination of the construct.

5.3.12 Helper rules



(a) Progress to the next instruction (b) Pop the next instruction from the stack

Figure 5.16: Next rules

When the instruction pointer points to an instruction which is marked as *finish*ed, one of these helper rules becomes active. These are responsible for progressing towards the next instruction. Either there is a *next* link (Figure 5.16a), which links towards the next instruction to execute. If it is present, the instruction pointer is moved to this instruction, and the phase is reset to *init* as it is the first time this construct is executed. In case no *next* link exists (Figure 5.16b), the next instruction is popped from the stack, together with its phase. This popping not only sets the instruction pointer, but also copies the saved phase, making it possible to progress where we left off.

5.3.13 Declare

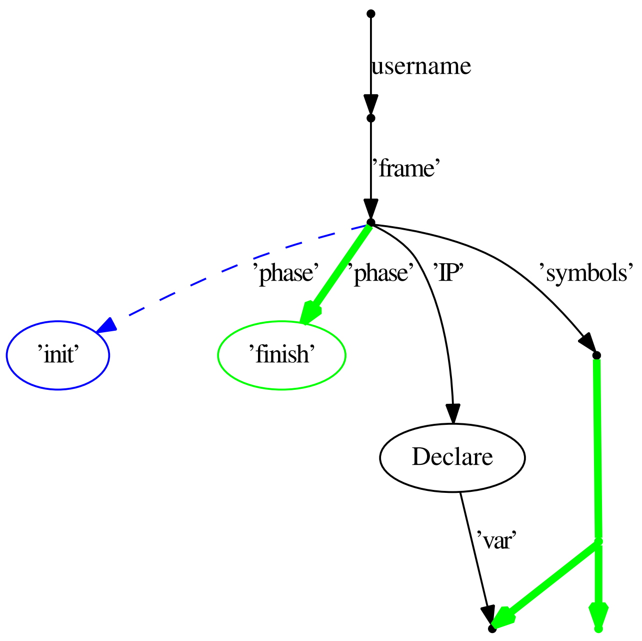


Figure 5.17: Declare instruction

The Declare instruction will add the specified node to the symbol table, so that it can be assigned a value, or read out. As the declare does not take a value, the default value of the node is just an empty node. Future instructions can use the node connected to the Declare instruction to reference to the variable.

5.3.14 Global

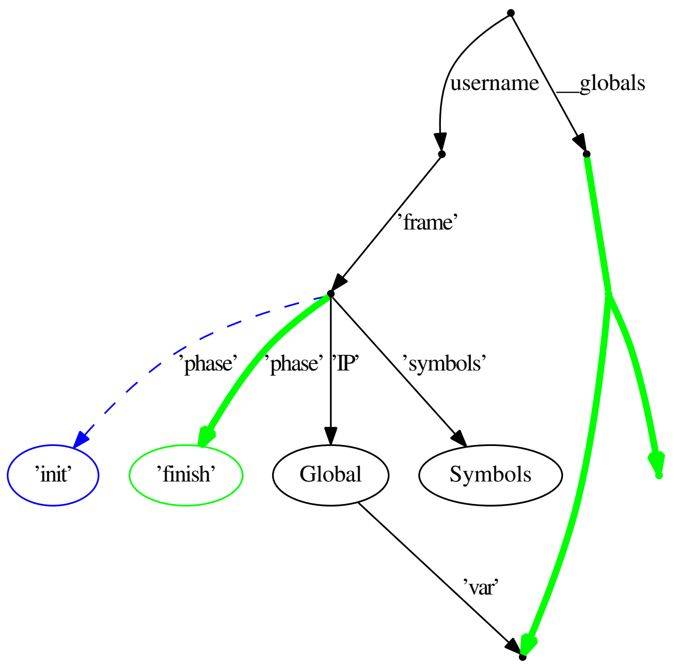


Figure 5.18: Global declare instruction

Apart from a declaration in the symbol table of the current user, it is also possible to declare it in the global namespace. This makes sure that other users can also find it and access the values. Its primary use will be function resolution though, as functions should be declared in a higher scope than the current scope. Nonetheless, it is possible to define everything else as a global too, making it accessible.

5.4 Primitive operations

As there are no special, built-in constructs for basic operations, such as mathematical operations, all of them have to map to a normal, user-level function. But these functions cannot implement the specified behaviour either, as the provided data values are MvS primitives. Such functions are primitive functions, which form the core of the MvK, and are hardcoded in the MvK implementation.

Primitive functions are hardcoded functions in the MvK, which get loaded like normal operations (*i.e.*, their parameters are evaluated and loaded on the stack). The execution of their body differs though, as it is executed without intermediate steps. As they cannot be written in Action Language, they do not have an implementation in the Action Language either. It is the MvK which recognizes that there is a primitive function available for the called function. If so, it calls the primitive instead of the (empty) body.

To comply with our axioms (Axiom IV: Model Everything), we need to model these functions explicitly. This can be done by taking the same approach as Squeak [3], where an interpreter is written in the interpreted language. Doing this, we can map the interpreted function (in the code being executed) to the primitive function of our used interpreter (in the implementation of the interpreter). Optionally, the interpreter could also be compiled, where these functions are then changed to primitive operations in the target language.

The operations in Table 5.2 and 5.3 need to be defined as a primitive by all Modelverse Kernel implementations, with the specified semantics. None of them are allowed to modify any of the incoming parameters. Semantics are given in simple Python code.

An MvK is free to implement additional functions as primitives, as long as each primitive instruction is guaranteed to terminate and does not violate the fairness between different users (Axiom X: Multi-User). Additionally, all additional functions need to have an equivalent implementation in Action Language for interoperability between different MvKs (Axiom XI: Interoperability). To enforce this fairness, and guarantee that all users have a fairly low response time, an upper bound is placed on the time allocated for such a primitive. If the operation times out, the operations done by the primitive are ignored and the function is interpreted as usual. The mandatory primitive operations should never time out due to their simplicity.

Modelled functions can therefore be compiled to new primitives for performance reasons (Axiom II: Scalability): they get mapped to native code, and they no longer need to update the execution context after every instruction. As the execution context is not updated, primitive operations cannot be debugged easily. For debugging, the user needs to be able to toggle an *interpreter-only* flag, which forces the Modelverse Kernel to execute in interpreter mode, bypassing all possible optimizations. This flag also requires the Modelverse Kernel to continuously update the execution context, as described in the previous sections. Execution of the primitives defined in Table 5.2 and 5.3 will still be through their hardcoded implementation though.

As almost everything is a function call, including mathematical operations, no order of operations is imposed, apart from the one in the function calls. Instead, the user is required to expand this to the correct function call. Most users, however, will use an MvI with a parsed concrete syntax, which can generate an automatically modified abstract syntax graph from this. Therefore, the user might still be able to write $d = a + b * c$, as long as the MvI expands this to $d = integer\_add(a, integer\_mul(b, c))$, taking into account the typing and order of evaluation during parsing. This offloads the work required for the implementation of a MvK.

Several primitive operations require some additional explanation:

- **float** operations only work on floats and not on integers, due to possible loss of accuracy. To get the desired results, explicit type conversions are required using the *cast* operations.
- **string** operations work on both strings and characters, as a character is a string of length 1.
- **type** will return the type of the provided element. This is possible as types of primitives are primitive data values too.
- **cast** operations are used to switch between types. Casts from a string will try to parse the result, whereas casting to a string will pretty-print the value. Boolean True is equal to integers or floats different from 0 or 0.0, respectively. Conversion from float to integer is rounded *down* if necessary.
- **create** operations are a one-to-one mapping with the MvS CRUD interface.
- **read_nr_(out/in)** returns the number of outgoing and incoming edges, respectively.
- **read_(out/in)** returns the specified outgoing or incoming edge, respectively.
- **read_dict** is a one-to-one mapping with the R_{dict} MvS CRUD operation, thus reading out from the dictionary based on value in the node.
- **read_dict** is a one-to-one mapping with the R_{dict_node} MvS CRUD operation, thus reading out from the dictionary based on the actual node.
- The **delete** operation will automatically determine the correct MvS delete operation to call.

Name	Parameters	Returns	Semantics
integer_addition	a : Integer; b : Integer	c : Integer	$c = a + b$
integer_subtraction	a : Integer; b : Integer	c : Integer	$c = a - b$
integer_multiplication	a : Integer; b : Integer	c : Integer	$c = a \times b$
integer_division	a : Integer; b : Integer	c : Integer	$c = a/b$
integer_eq	a : Integer; b : Integer	c : Bool	$c = a == b$
integer_neq	a : Integer; b : Integer	c : Bool	$c = a \neq b$
integer_lt	a : Integer; b : Integer	c : Bool	$c = a < b$
integer_lte	a : Integer; b : Integer	c : Bool	$c = a \leq b$
integer_gt	a : Integer; b : Integer	c : Bool	$c = a > b$
integer_gte	a : Integer; b : Integer	c : Bool	$c = a \geq b$
integer_neg	a : Integer	c : Bool	$c = -a$
float_addition	a : Float; b : Float	c : Float	$c = a + b$
float_subtraction	a : Float; b : Float	c : Float	$c = a - b$
float_multiplication	a : Float; b : Float	c : Float	$c = a \times b$
float_division	a : Float; b : Float	c : Float	$c = a/b$
float_eq	a : Float; b : Float	c : Bool	$c = a == b$
float_neq	a : Float; b : Float	c : Bool	$c = a \neq b$
float_lt	a : Float; b : Float	c : Bool	$c = a < b$
float_lte	a : Float; b : Float	c : Bool	$c = a \leq b$
float_gt	a : Float; b : Float	c : Bool	$c = a > b$
float_gte	a : Float; b : Float	c : Bool	$c = a \geq b$
float_neg	a : Float	c : Bool	$c = -a$
bool_and	a : Bool; b : Bool	c : Bool	$c = a \wedge b$
bool_or	a : Bool; b : Bool	c : Bool	$c = a \vee b$
bool_not	a : Bool	c : Bool	$c = \neg a$
list_append	a : Element; b : Element	a : Element	$a+ = b$
list_insert	a : Element; b : Element; c : Integer	a : Element	$a.insert(b, c)$
list_delete	a : Element; b : Integer	a : Element	$a = a.pop(b)$
list_len	a : Element	b : Integer	$b = len(a)$
dict_add	a : Element; b : Element, c : Element	a : Element	$a[b] = c$
dict_delete	a : Element; b : Element	a : Element	$delete\ a[b]$
dict_read	a : Element; b : Value	c : Element	$c = a[b]$
dict_read_node	a : Element; b : Element	c : Element	$c = a[b]$
dict_len	a : Element	b : Integer	$b = len(a)$
dict_in	a : Element; b : Value	c : Boolean	$c = bina$
dict_in_node	a : Element; b : Element	c : Boolean	$c = bina$
string_join	a : String; b : String	c : String	$c = a.b$
string_get	a : String; b : Integer	c : String	$c = a[b]$
string_substr	a : String; b : Integer; c : Integer	d : String	$d = a[b : c]$
string_len	a : String	b : Integer	$b = len(a)$
set_add	a : Element; b : Element	a : Element	$a.add(b)$
set_pop	a : Element	b : Element	$b = a.pop()$
set_remove	a : Element; b : Element	a : Element	$a.remove(b)$
set_in	a : Element; b : Element	c : Boolean	$c = bina$
action_eq	a : Action; b : Action	c : Bool	$c = a == b$
action_neq	a : Action; b : Action	c : Bool	$c = a \neq b$
type_eq	a : TypeType; b : TypeType	c : Bool	$c = a == b$
type_neq	a : TypeType; b : TypeType	c : Bool	$c = a \neq b$
typeof	a : Element	b : TypeType	$b = type(a)$

Table 5.2: Primitive functions modifying primitive datavalues. If a Value is taken or returned, this refers to the value of the returned node.

Name	Parameters	Returns	Semantics
cast_i2f	a : Integer	b : Float	$b = float(a)$
cast_i2s	a : Integer	b : String	$b = str(a)$
cast_i2b	a : Integer	b : Bool	$b = bool(a)$
cast_f2i	a : Float	b : Integer	$b = int(a)$
cast_f2s	a : Float	b : String	$b = str(a)$
cast_f2b	a : Float	b : Bool	$b = bool(a)$
cast_s2i	a : String	b : Integer	$b = int(a)$
cast_s2f	a : String	b : Float	$b = float(a)$
cast_s2b	a : String	b : Bool	$b = bool(a)$
cast_b2i	a : Bool	b : Integer	$b = int(a)$
cast_b2f	a : Bool	b : Float	$b = float(a)$
cast_b2s	a : Bool	b : String	$b = str(a)$
cast_e2s	a : Element	b : String	$b = str(a)$
create_node	—	a : Element	create node and return ID
create_edge	a : Element; b : Element	c : Edge	create edge from a to b and return ID
create_value	a : Value	b : Element	create node with value a and return ID
is_edge	a : Element	b : Boolean	return whether a is an edge or not
read_nr_out	a : Element	b : Integer	return number of outgoing links from a
read_out	a : Element; b : Integer	c : Element	return the bth element which has an outgoing link from a
read_nr_in	a : Element	b : Integer	return number of incoming links from a
read_in	a : Element; b : Integer	c : Element	return the bth element which has an incoming link from a
read_edge_src	a : Edge	b : Element	return the source of edge a
read_edge_dst	a : Edge	b : Element	return the destination of edge a
delete_element	a : Element	a : Boolean	delete element a
element_eq	a : Element; b : Element	c : Boolean	return whether or not a and b are exactly equal
deserialize	a : String	b : Element	merge serialized graph with current and return initial node
import_node	a : String	b : Element	import a previously exported node
export_node	a : String; b : Element	b : Element	export a node so it can be imported

Table 5.3: Lower-level primitive functions to implement. If a Value is taken or returned, this refers to the value of the returned node.

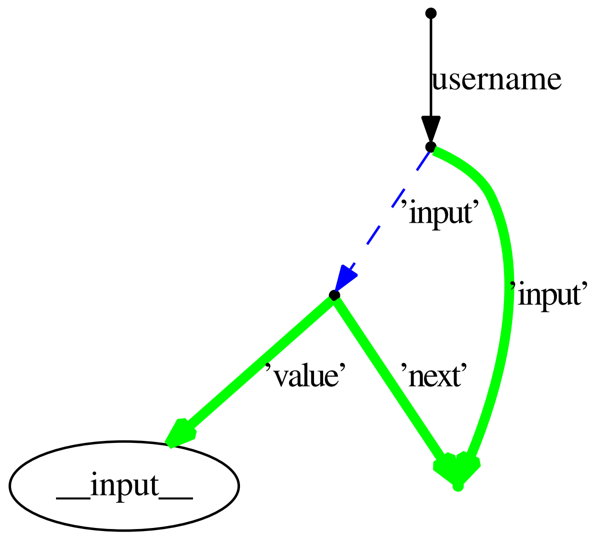
5.5 Interface

Since the MvK is not an autonomous process, it requires input from the user, and needs to forward output to the user when required. Therefore, an interface towards the MvI is required, which offers only two functions: add something to the input queue, and pop something from the output queue.

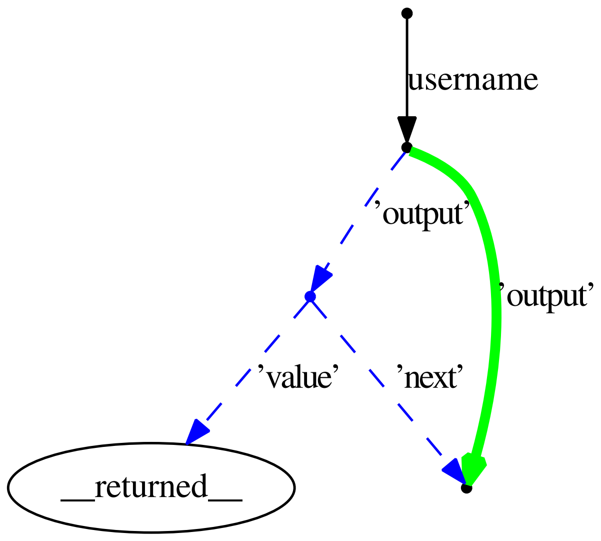
This interface is sufficient to execute all operations, as the input value can be any element in the modelverse, for example a function signature or name to resolve and subsequently execute. While this makes the interface very minimal, it pushes all API definitions to the MvK itself, thus explicitly modelling parts that were normally hardcoded.

Another advantage of this very versatile API, is that the MvK can be customized per-user. The running process of the user will just have to function as the API itself, and process all incoming messages. It also makes sure that all desired functionality is present, as users can manually implement it if necessary.

We now formalize the behaviour of these two functions (`set_input` and `get_output`) just like the execution rules. It should be noted however, that these rules are not executed when they are applicable, but only when they are invoked through the API. The rules also reference elements that are passed to the invoking API call, and returns the marked node.



(a) API rule for input processing



(b) API rule for output processing

Figure 5.19: API rules

6 Modelverse Interface

6.1 Browser

As the Modelverse uses ordinary XML/HTTP requests, it is possible to easily transfer data through normal web clients, such as internet browsers. Since these clients are not built with the Modelverse in mind, users will have to use the raw API of the Modelverse Kernel (*i.e.*, using `set_input` and `get_output`).

6.2 Serialized Graph

The most efficient, though also most low-level, way of transferring a model to the Modelverse, is through the serialisation of a graph structure. This graph, following a very minimal description format, will subsequently be merged into the current Modelverse graph. After serialization, the entry point to the graph is returned.

This approach is very efficient, as it constitutes a single call to the Modelverse with the complete graph. All serialization and deserialization will happen in pure Python code (thus not explicitly modelled!), so basically everything is possible.

The problem with this approach is that the interface needs to know the internal representation of models inside of the Modelverse. All connections need to be made manually, and the serialization format also needs to be known. It is therefore not recommended to use this interface, unless no other option is possible (*i.e.*, during bootstrapping).

6.3 HUTN for Action Language

The more elegant way of sending action code to the Modelverse, is by using the explicitly modelled constructors. These constructors are simple action code that was previously loaded inside the Modelverse, which interprets the requests send to it as creation requests. While this is much less efficient, models can be created on-the-fly (without first creating the complete graph in the interface), and follows a standardized interface. Future changes to the internal representation of models will not affect these calls, as the actual creation of the model happens in the Modelverse itself. Users need thus only know Table **??**, which contains a list of all understood construction requests.

Apart from hiding basic complexity (the names of the links), it also has simple requests to make more difficult constructs, such as function declarations or function calls. Furthermore, all complexity is hidden from the user, as all identifiers are kept in the Modelverse. Only those identifiers that need to be passed (*i.e.*, variable declarations), will be provided to the interface.

This is the recommended interface to use when transferring models to the Modelverse.

6.4 HUTN for Models

instruction	parameter	type
if	condition	instruction
	then	instruction
	else?	instruction
	next?	instruction
while	condition	instruction
	body	instruction
	next?	instruction
access	lvalue	instruction
resolve	variable	variable
assign	lvalue	instruction
	rvalue	instruction
	next?	instruction
call	function	instruction
	nrParams	integer
	name#	string
	value#	instruction
	next?	instruction
return	value?	instruction
const	value	anything
declare	variable!	variable
global	variable!	variable
output	value	instruction
	next?	instruction
input		
deref	link	string
funcdef	variable	variable
	nrParams	integer
	name#	string
	formal!#	variable
	body	instruction
	next?	instruction
break		
continue		

Table 6.1: List of constructor keywords and parameters they take (in order). Questionmark after a parameter means that you first need to pass true/false for whether or not the element exists. Exclamation mark indicates that this is output at this place. A hash means that it occurs as often as previously specified.

7 Network communication

7.1 Motivation

7.2 Modelverse State

7.2.1 Interface

The XML/HTTPRequest back-end of the MvS will simply host an HTTP server, which responds to POST requests. The reply of the server is again encoded in the same format as the POST request.

All requests should be send via POST, and contain the following two parameters:

- **op**: this indicates which operation to execute on the MvS.
- **params**: contains the parameters for the function, encoded in JSON format. While we require JSON encoding, the data can never be complex due to the simple signature of the supported operations. This parameter should always be a list of the parameters to pass. If there is only a single parameter, a list with a single element is still required.

The operations all use coding, to reduce the amount of data that needs to be transfered. Table 7.1 shows the mapping between the operation and the formalized function name.

Listing 7.1: Example request and reply

```
Request: op=RE&params=[1]
Reply:   data=[[2, 3], 100]
```

An example request, and corresponding reply, is shown in Listing 7.1, where an edge with identifier 1 is read. The reply indicates that the request was succesful (statuscode 100), and the returnvalue indicates that edge 1 goes from element 2 to element 3.

Note that the $\mathcal{G}$ parts of the request and reply, as were formalized previously, are not included. This is because the MvS itself is the instance of $\mathcal{G}$ being modified.

Sockets are kept open until explicitly closed, so it is possible to reuse a single socket for every request. It is also possible to send a request before the previous request is handled of. In that case, the order of the replies will be the same as of the requests.

7.2.2 Statechart

7.3 Modelverse Kernel

7.3.1 Interface

Communication with the Modelverse Kernel is very similar to the communication with the Modelverse State. The Modelverse accepts two different operations: `set_input` and `get_output`. Data is to be send as a POST request, and has to consist of the following fields:

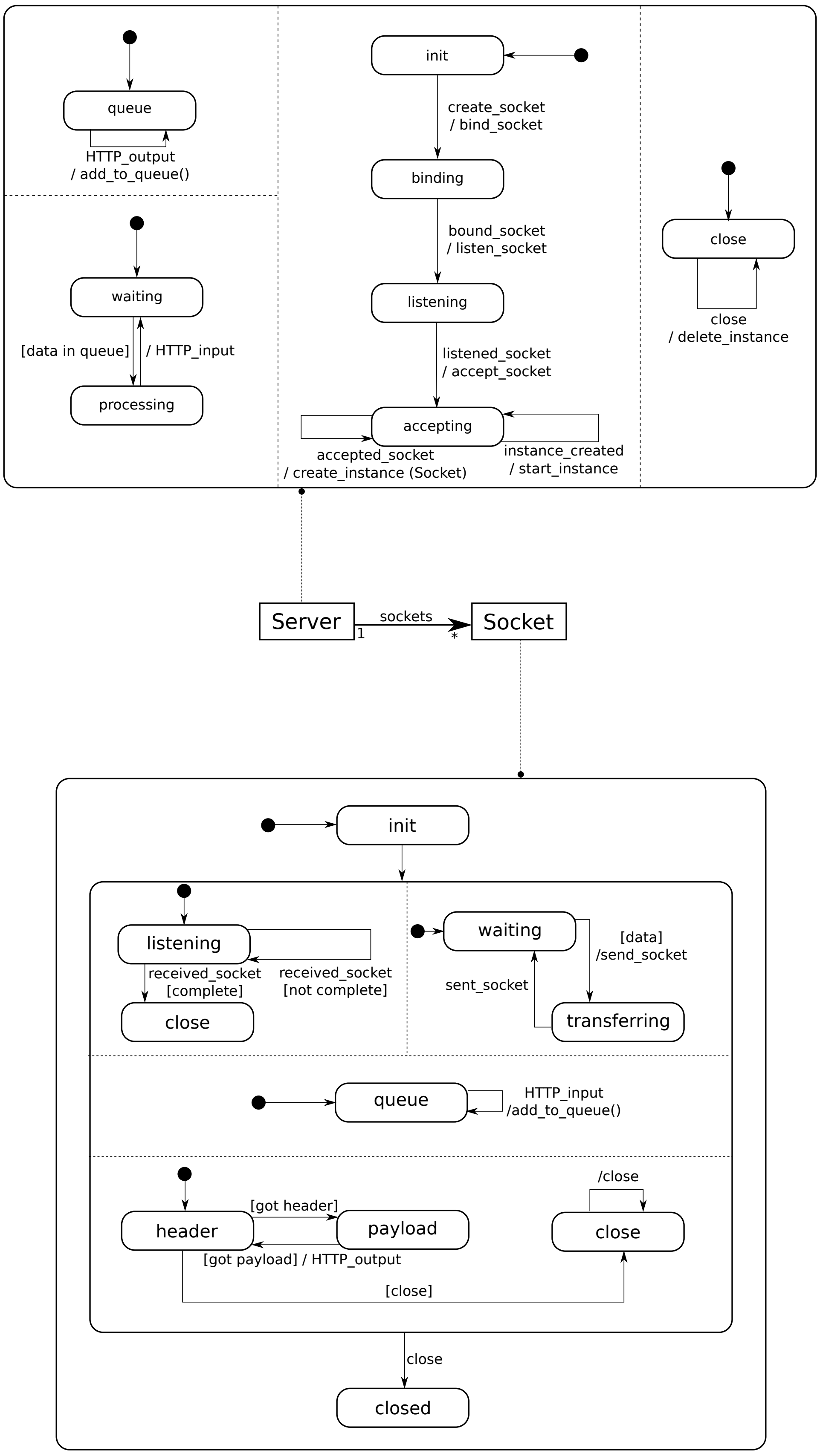


Figure 7.1: Modelverse State server statechart

Operation	Formal function
CN	create_node
CE	create_edge
CNV	create_nodevalue
CD	create_dict
RV	read_value
RO	read_outgoing
RI	read_incoming
RE	read_edge
RD	read_dict
RDN	read_dict_node
RDE	read_dict_edge
RRD	read_reverse_dict
RR	read_root
RDK	read_dict_keys
DE	delete_edge
DN	delete_node

Table 7.1: Mapping between operations and formalized function name

1. **op**: the operation to perform. It can be either ”set_input” or ”get_output”. Depending on the value of this entry, some additional elements need to be present in the request.
2. **username**: the name of the user whose input or output queue is modified. Always present for both operations.
3. **element_type**: how to interpret the value parameter. It is either ”R”, to indicate that the value parameter is a reference, and therefore an element identifier. The other option is ”V”, to indicate that the value parameter is a JSON encoded value. Only present if the operation is set_input.
4. **value**: the actual parameter to the operation. Its interpretation is given by the element_type operation. If it has to be interpreted as a value, it needs to be an instance of a primitive for the MvS. Only present if the operation is set_input.

For both requests, a reply will be returned containing an *id* and *value* entry.

For the `set_input`, the *id* and *value* are a status code and human-readable description. Generally, giving input should always succeed, resulting in *id* 100 and *value* `success`.

For the `get_output`, the *id* will be the identifier of the node that is to be output. The *value* is the value of the node with the provided identifier. Getting output is a blocking call, so the request will stay open until input is actually generated. As soon as the output is generated, it will be sent out.

An example request and reply is shown in Listing 7.2 and 7.3, for set_input, and Listing **??** and **??**, for get_output.

Listing 7.2: Example: create new user

```
Request: op=set_input&username=user_manager&element_type=V&value="user_1"
Reply:   id=100&value="success"
```

Listing 7.3: Example: input element ID 15 for user

```
Request: op=set_input&username=user_1&element_type=R&value=15
Reply:   id=100&value="success"
```

Listing 7.4: Example: read output valuelabel

```
Request: op=get_output&username=user_1
Reply:   id=123&value="node_value"
```

7.3.2 Statechart

7.4 Modelverse Interface

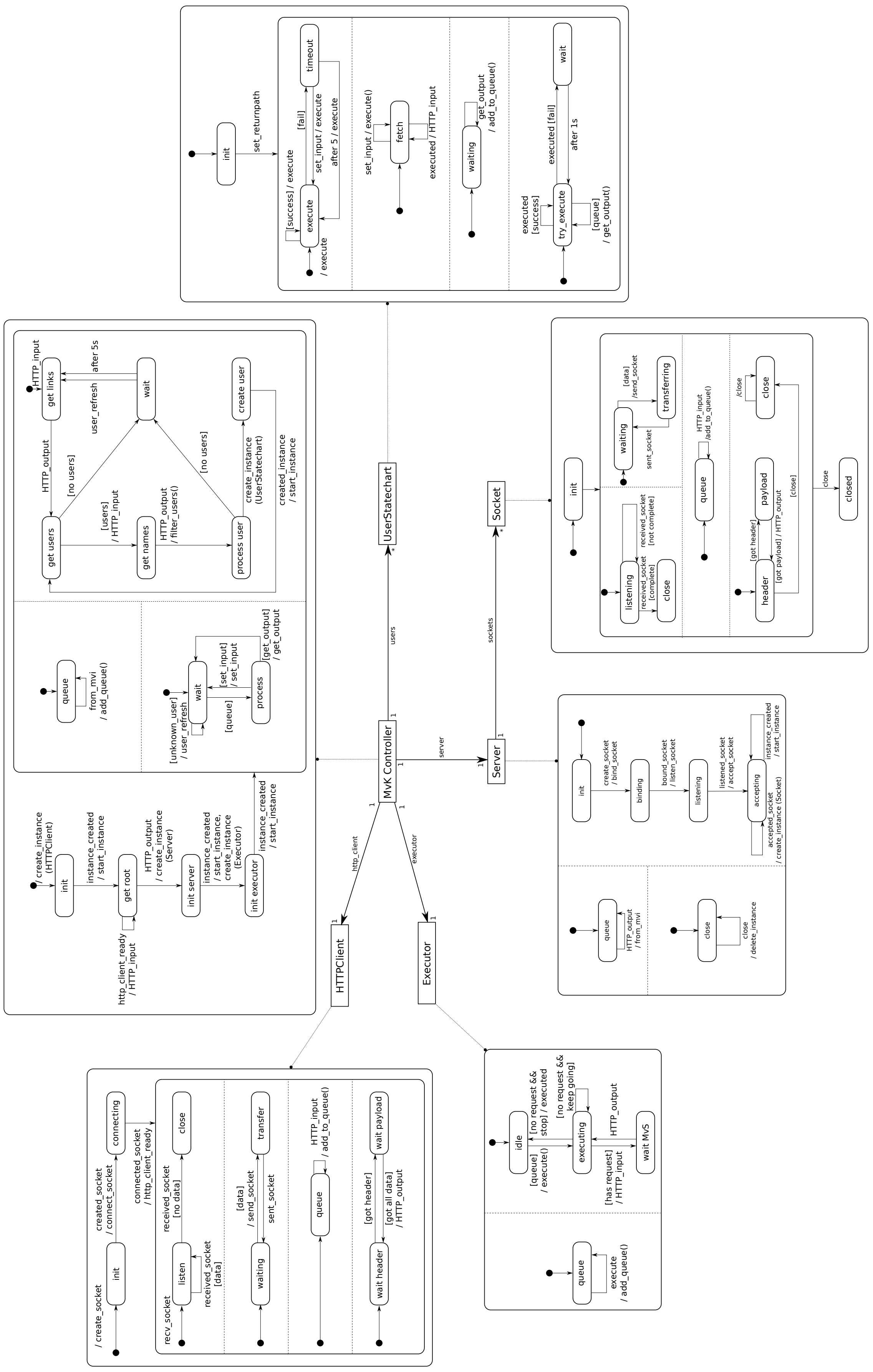


Figure 7.2: Modelverse Kernel server statechart

8 Conformance

Model management operations (*e.g.*, conformance checking, or versioning) frequently act upon both the model and the metamodel, or should be applicable on all models, independently of their metamodel. As such, they are often in conflict with the principle of strict metamodelling. We try to find a balance between strict metamodelling (Axiom V: Human Interaction), and the principle of modelling everything explicitly (Axiom IV: Model Everything).

By introducing multiple definitions of conformance, we can keep strict metamodelling while still implementing such model management functions. The basic idea is to allow a single model to conform to multiple metamodels. The conceptual graph, representing the model, is interpreted depending on the metamodel being used. Examples of metamodels might be a domain-specific metamodel (*e.g.*, a Petri Net metamodel), or a more physically-oriented metamodel (*e.g.*, a Graph metamodel). Depending on the interpretation given to the levels, different level hierarchies are constructed. It is these level hierarchies that impose the restrictions on strict metamodelling.

Fig. 8.1 presents some different notions of conformance that can be devised on the Modelverse. While the amount of conformance relations can vary, each model will have a mapping to the PTM, which is required to physically represent the model. And it will have a mapping to the Linguistic Type Model (LTM), using conformance$_\perp$. This relates to Axiom VII: Multi-View, as a single model can be seen from different views, and relates to Axiom XI: Interoperability, as it allows for the uniform representation of all data.

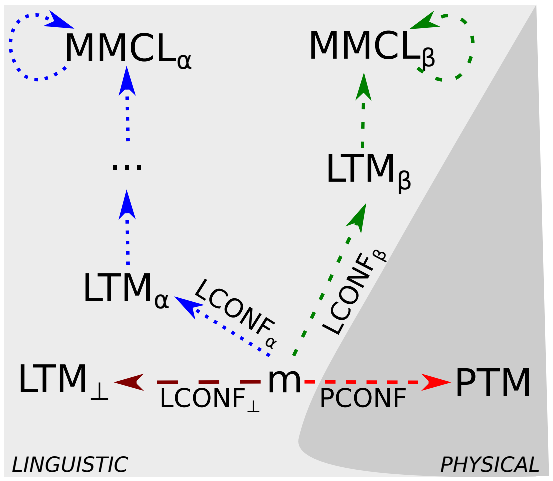


Figure 8.1: Different conformance relations

8.1 Graph conformance

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 defined CRUD operations constrain the result to a well-formed graph, all models in the Modelverse conform to this metamodel by construction. Every model represented in the Modelverse is conceptually representable as a graph. Knowing this, the complete Modelverse can be flattened to a single level, which conforms to the graph formalism. Within this single level, all operations and links between elements are non-level crossing, and are therefore correctly typed (by the graph metamodel). Note however, that these methods are unable to guarantee conformance to any linguistic metamodel, apart from the graph metamodel. As such, all multi-formalism models can also be represented using this single metamodel, thus partially addressing Axiom VIII: Multi-Formalismand Axiom IX: Multi-Abstraction. Multiple users (Axiom X: Multi-User) can also use this view to collaborate on a single model, while having different interpretations of it.

8.2 Linguistic conformance

Finally there is the linguistic conformance between the model and the metamodel, which is necessary to complete the support for our axioms (Axiom V: Human Interaction). It is the highest level, and offers the most features to the user, but is also the most fragile. Linguistic conformance cannot be guaranteed at all, and requires continuous checking to make sure it is enforced for the desired model and metamodel. In contrast, conformance$_\perp$ was guaranteed by design.

Because a conformance$_L$ view is only a specific view on a model, a single model can conform to multiple metamodels. The function $conforms : \mathcal{G} \times \mathcal{G} \times 2^{IDS \to IDS} \to \mathbb{B}$ is defined to determine linguistic conformance, and can be implemented by the user. It takes three parameters: two graphs — a model and a metamodel, both subgraphs of the MvS graph — and a mapping between them. This mapping encapsulates all typing information, thus typing is completely separated from the model and metamodels. Since multiple mappings can be stored, multiple typing relations are supported. During syntax-directed editing, a mapping will be constructed (and used) with the information provided by the user. Retyping can be done by modifying the mapping, and checking conformance afterwards.

We define a possible conformance$_L$ relation, to be seen as an example of our approach. This relation bases itself on the top level model: the Model at the MetaCircular Level (MMCL). In this model, we have three basic elements: a *Class* (mapped to nodes), an *Association* (connecting classes; mapped to edges), and *Inheritance* between classes (mapping to the union type). Using *Inheritance*, the *Association* becomes a special kind of *Class* in our MMCL. We call our example conformance relation *conformance*$_\alpha$ to indicate that it is one of many possible implementations.

First, we define a subfunction which defines transitive closure of inheritance links, where $A \leq B$ means that A is a (possibly indirect) subclass of B. $A \xrightarrow{\triangleright} B$ means that there is an association, typed by the *Inheritance* link, from A to B.

$$\begin{aligned} A \xrightarrow{\triangleright} B &\Rightarrow A \leq B \\ A \leq C \wedge C \leq B &\Rightarrow A \leq B \\ A == B &\Rightarrow A \leq B \end{aligned}$$

A conformance relation for the primitive elements is defined, constraining the provided map.

$$conforms : (N \cup E) \times (N \cup E) \times 2^{IDS \times IDS} \to \mathbb{B}$$

$$conforms(x,y,m) = \begin{cases} N_T(N_V(x)) == N_V(y) & if\ x,y \in dom(N_V) \wedge (x,y) \in m \\ True & if\ x,y \in N \wedge x,y \notin dom(N_V) \wedge (x,y) \in m \\ conforms(x_s,y_s,m) \wedge conforms(x_t,y_t,m) & if\ (x,y),(x_s,y_s),(x_t,y_t) \in m \wedge (x_s,x,x_t),(y_s,y,y_t) \in E \\ False & else \end{cases}$$

The first line is for nodes with a primitive value: a node x conforms to a node y if both nodes have a value, with the type of the value of node x being the value of node y. The second line is for nodes without a primitive value: a node x conforms to a node y if such a mapping exists in the provided mapping. Neither node is allowed to have a primitive value. The third line is for edges: an edge x conforms to an edge y if their sources and targets conform to each other. It is thus basically a recursive call. However, there is no possibility for an infinite loop, because of our restriction on the IDs of edges: the source and target ID are always smaller than the ID of the edge. The final line is for all other cases (*e.g.*, comparing nodes to edges, or primitives to non-primitives), in which case there is no conformance possible.

Finally, $conforms_{\alpha,G} : \mathcal{G} \times \mathcal{G} \times 2^{IDS \to IDS} \to \mathbb{B}$ is the actual conformance function being called. It tries to find a mapping between the specified model and metamodel, for which the conforms function holds.

Listing 8.1: HUTN$_\perp$ construction of the MMCL

```
Node Class()
Value Type(Type)
Value String(String)
Edge Attribute_ (Class, Type)
Edge AttributeAttrs (Attribute_, Type)
Edge Attribute (Class, Type)
Edge Name (Attribute, String)
Edge Association (Class, Class)
Edge Inheritance (Class, Class)
Edge inherit_association (Association, Class)
Edge inherit_attribute (Attribute_, Class)
```

$$
\begin{gathered}
conforms_{\alpha,G}(M, MM, map) = True \\
\Leftrightarrow \\
map' = \{(a,b) \mid a \in IDS_M, b \in IDS_{MM}\} \\
\forall n \in N_M : \exists n' \in N_{MM}.conforms(n, n', map') \\
\forall e \in IDS_{E,M} : \exists e' \in IDS_{E,MM}.conforms(e, e', map') \\
\forall (a_i, b_i), (a_i, b_j) \in map' : (b_i \leq b_j) \\
\forall (a_i, b_i) \in map' : (a_i \rightarrow b_k) \in map \wedge b_k \leq b_i \\
\forall a, b \in IDS : a \leq b \wedge b \leq a \Rightarrow a == b
\end{gathered}
$$

A map is generated which contains all possible mappings between the model and metamodel. This map is then constrained by enforcing a mapping for the nodes and edges. Source and target of the edges are recursively checked for conformance using the mapping. We finally prune the set of possible mappings by only keeping a single type mapping for every node, with the exception being subclasses. Finally, this mapping is pruned to a function by keeping the most specific (or a more specific) subclass of all present mappings.

Just like a model can conform to multiple metamodels, a model can also conform to the same metamodel multiple times, with different mappings.

Using this function, we can now check whether a model conforms to a specified metamodel, using a specified mapping. It is also possible to generate a set of all possible mappings between a model and a metamodel.

$$
\begin{gathered}
MAP = 2^{IDS \rightarrow IDS} \\
mappings_{\alpha,G} : \mathcal{G} \times \mathcal{G} \rightarrow 2^{MAP} \\
mappings_{\alpha,G}(M, MM) = s \\
s = \{map \in MAP \mid conforms_{\alpha,G}(M, MM, map)\}
\end{gathered}
$$

8.3 MMCL

We present an encoding of our MMCL, in Listing 8.1, using the HUTN language respecting conformance$_\perp$. The action code in this language is translated to an abstract syntax graph in the Modelverse, by a HUTN compiler. The HUTN compiler lives in a Modelverse Interface (MvI).

Using this MMCL, we can now re-encode it, as in Listing 8.2, now using the HUTN language with conformance$_L$. Alternatively, it is possible to directly use the definition in Listing 8.2, as elements can directly be typed by themselves in the Modelverse.

Finally, we encode our conformance checking algorithm, in Listing 8.3, using the HUTN action language. With this example we show (1) an example of modelling, as the action code is a model, and thus an element of the Modelverse; (2) an example of our action code; (3) the possibility for reflection and introspection, as the conformance check can also run on itself, to check whether or not it conforms to some kind of metamodel; and (4) the possibility for metamodelling, as type hierarchies can be built using the provided conformance function.

Listing 8.3: HUTN conformance check algorithm

Listing 8.2: HUTN$_L$ construction of the MMCL

```
Class Class()
Type Type(Type)
Type String(String)
Attribute_ Attribute_ (Class, Type)
Attribute_ AttributeAttrs (Attribute_, Type)
Attribute_ Attribute (Class, Type)
AttributeAttrs Name (Attribute, String)
Association Association (Class, Class)
Association Inheritance (Class, Class)
Inheritance (Association, Class)
Inheritance (Attribute_, Class)
```

```
include "primitives.al"

Element function set_copy(elem : Element):
    Element result
    Integer counter
    Integer max

    result = create_node()

    // Expand the provided list by including all elements that need to be checked
    counter = 0
    max = read_nr_out(elem)
    while (integer_lt(counter, max)):
        set_add(result, read_edge_dst(read_out(elem, counter)))
        counter = integer_addition(counter, 1)

    return result

Boolean function is_subclass_of(superclass : Element, subclass : Element, types : Element, inheritance_
    Integer counter_iso
    Integer i
    Element edge
    Element destination

    if (element_eq(superclass, subclass)):
        return True

    counter_iso = read_nr_out(subclass)
    i = 0
    while (integer_lt(i, counter_iso)):
        edge = read_out(subclass, i)
        if (dict_in_node(types, edge)):
            if (element_eq(dict_read_node(types, edge), inheritance_link)):
                // It is an inheritance edge, so follow it to its destination
                destination = read_edge_dst(edge)
                if (is_subclass_of(superclass, destination, types, inheritance_link)):
                    return True
        i = integer_addition(i, 1)

    // No link seems to have been found, so it is False
    return False

String function conformance_scd(model : Element, extra : Element):
    // Initialization
    Element work_conf
    Element model_src
    Element metamodel_src
    Element model_dst
    Element metamodel_dst
    Element models
    Element metamodels
    models = set_copy(model)

    Element typing
    typing = dict_read(extra, "typing")
    metamodels = set_copy(dict_read_node(typing, model))
    Element inheritance
    inheritance = dict_read(extra, "inheritance")

    // Iterate over all model elements and check if they are typed (in "typing") and their type is in t
    while (integer_gt(dict_len(models), 0)):
        work_conf = set_pop(models)
        // Basic check: does the element have a type
        if (bool_not(dict_in_node(typing, work_conf))):
            return string_join("Model has no type specified: ", cast_e2s(work_conf))

        // Basic check: is the type of the element part of the metamodel
        if (bool_not(set_in(metamodels, dict_read_node(typing, work_conf)))):
            return string_join("Type of element not in specified metamodel: ", cast_e2s(work_conf))

        // Basic check: type of the value agrees with the actual type
        // this is always checked, as it falls back to a sane default for non-values
        if (bool_not(type_eq(dict_read_node(typing, work_conf), typeof(work_conf)))):
            output(dict_read_node(typing, work_conf))
            output(typeof(work_conf))
            return string_join("Primitive type does not agree with actual type: ", cast_e2s(work_conf))

        // For edges only: check whether the source is typed according to the metamodel
        if (is_edge(work_conf)):
            model_src = read_edge_src(work_conf)
            metamodel_src = read_edge_src(dict_read_node(typing, work_conf))
            if (bool_not(is_subclass_of(metamodel_src, dict_read_node(typing, model_src), typing, inher
                return string_join("Source of model edge not typed by source of type: ", cast_e2s(work_

        // For edges only: check whether the destination is typed according to the metamodel
        if (is_edge(work_conf)):
            model_dst = read_edge_dst(work_conf)
            metamodel_dst = read_edge_dst(dict_read_node(typing, work_conf))
            if (bool_not(is_subclass_of(metamodel_dst, dict_read_node(typing, model_dst), typing, inher
                return string_join("Destination of model edge not typed by destination of type: ", cast

    return "OK"
```

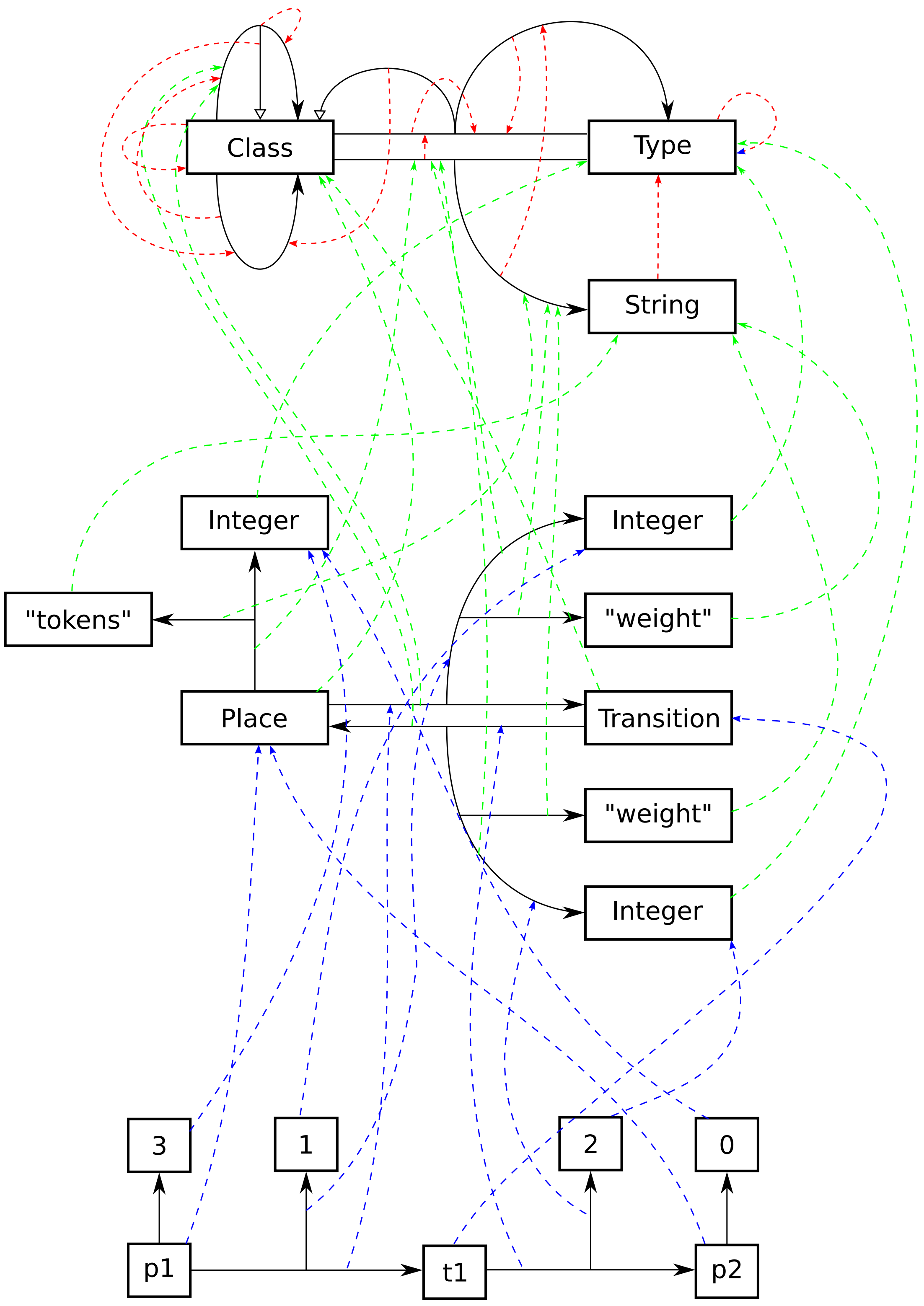


Figure 8.2: Conformance example for a petri net.

9 Practical information

This chapter describes how to execute and use our proof of concept implementation of the Modelverse. This implementation follows the previously defined interface, and is implemented in Python. Other implementations are possible, since each part of the service runs separately and they communicate through the use of sockets. As such, more efficient implementations in compiled programming languages (*e.g.*, C++) are possible.

9.1 Requirements

The proof of concept implementation uses Python 2.7. As all aspects are explicitly modelled, this platform is the only dependency. For the testing framework, `py.test` is recommended, though it is compatible with the default `unittest` module of Python.

All mentioned scripts are developed primarily for Linux, using shell scripts. Often though, Windows batch scripts are provided which should have identical behaviour.

9.2 Test suite

To run the tests, it suffices to execute `py.test` in the folder of the project. Since the Modelverse project consists of several subprojects (Modelverse State, Modelverse Kernel, and Modelverse Interface), it should be invoked in each folder seperately. For this, a script `run_tests.sh` is provided.

Additionally, some "integration" tests are provided, which set up a complete Modelverse process and accesses it through the usual Modelverse Interface API. These tests are also ran using the `run_tests.sh` script.

9.3 Running the Modelverse

Manually running the Modelverse happens, again, through the invocation of the script `run_local_modelverse.sh`. This script takes a single parameter: a file containing the initial state of the Modelverse, called `bootstrap.m`.

Note that this section uses the `run_local_modelverse.sh` script, instead of `run_modelverse.sh`. The former contains an optimized implementation of the Modelverse, which directly couples the Modelverse Kernel and Modelverse State, instead of having them communicate through sockets. Both situations will work, though the former is much more efficient at this time. No modifications to either component is necessary for this, as it only changes a small part of the network communication between them (directly coupling both components).

This script will first compile the necessary Modelverse wrapper statechart, and afterwards executes it. Now that the Modelverse is running, by default on port 8001, it can be accessed through XML/HTTP requests.

9.4 Bootstrap file

The bootstrap file contains the initial state of the Modelverse upon startup. It contains essential constructs, such as the primitives (*e.g.*, `integer_addition`, `create_node`), and the initial user (*user_manager*, for generating further users). While it should normally not be changed, this initial content can be automatically generated through the `bootstrap/bootstrap.py` script. The script contains a basic configuration for determining which primitives need to be loaded, and what the initial structure of the Modelverse should be upon creation.

By default, the bootstrap file initializes each user with code to deserialize an encoded string to a graph that will be merged in the Modelverse State. After merging, the provided graph will be executed. If the provided code returns True, a deserialize call will be invoked again, otherwise the user stops execution.

9.5 XML/HTTP requests

As the Modelverse listens for XML/HTTP requests, every possible XML/HTTP request-capable client can be used. In the limit, this can be even a simple command line tool, such as *curl*. An example *curl* invocation to create a new user called "test" is `curl http://localhost:8001 -d "op=set_input&username=user_manager&element_type=V&value="test""`. To get output of the user, the *curl* invocation is `curl http://localhost:8001 -d "op=get_output&username=test&element_type=V&value="`.

9.6 Compiling with HUTN

Manually using the XML/HTTP interface is clearly not desirable for end-users. As such, an MvI is needed to hide this complexity from the users. An example MvI, in the form of a HUTN compiler, is provided and will be introduced now. The compiler can be invoked through the script `./execute_as.sh`. It takes two parameters: the name of the user that needs to be created, and a file containing the HUTN to execute. The script automatically creates the specified user, compiles the provided file, uploads it, and finally executes it.

As mentioned previously, the default bootstrap file waits for an encoded string that contains a graph. This graph is then executed and will further act as the service that is being executed. This way, it is actually possible to define every possible interface by explicitly modelling it.

9.7 Examples

Finally, we introduce some simple examples that show how the HUTN compiler can be used and what the results are. More examples are provided in the test suite.

9.7.1 Simple Action Language Services

First, to show that every kind of service can be modelled explicitly, we define a simple arithmetic service, shown in Listing 9.1. This service will continuously wait for input, and respond with the factorial of this number. The example essentially consists of three parts:

1. *Imports*: as everything is explicitly modelled, even the primitive operations need to be explicitly loaded. This can be done by including the file "primitives.al". More specific imports are also possible, like "integer.al", "float.al", etc.
2. *Code* The actual algorithm is stored here, and is written in a minimal action language syntax. The core of the algorithm is very similar to how the implementation would be in another implementation langauge. Most notably, there is currently no support for operators, so each part has to be explicitly invoked as a function.
3. *Main loop* As the code defines its own interface, a main loop will also be required for our example. This main loop is just a simple inifite while loop, which takes input, passes it to the defined algorithm, and outputs the result. In more complex situations, this main loop can contain the actual decoding of the incoming message.

Listing 9.1: Example factorial service.

```
include "integer.al"

Integer function factorial(n : Integer):
    if(integer_lte(n, 1)):
        return 1
    else:
        return integer_multiplication(n, factorial(integer_subtraction(n, 1)))

while(True):
    output(factorial(input()))
```

To have the Modelverse execute this piece of code for a specific user, the `execute_as.sh` script can be invoked as follows: `./execute_as.sh test factorial.al`. This will initiate compilation of the action language code to a graph representation, which is subsequently uploaded to the Modelverse. Now, the Modelverse will start to execute the provided graph, and blocks for input.

A user can now provide input to this method, by sending the input to the previously defined user. This can be done as follows: `curl http://localhost:8001 -d "op=set_input&username=test&element_type=V&value=5"`. After which the Modelverse will start to compute this value. Immediately after, the output can be requested (as it will block anyway) as follows: `curl http://localhost:8001 -d "op=get_output&username=test&element_type=V&value="`. This request will eventually return with a response similar to this: `id=12345&value=120`. Note that the id might be different, though the value should be identical.

9.7.2 Model Conformance Checks

A more complex example is closer to the problem the Modelverse tries to solve: modelling operations. Special syntax is provided to create models. After the models are created, they can be used just like any other element of the Modelverse. As such, there is no fundamental distinction between a user-made model, and a built-in primitive.

The example in Listing 9.3 shows how three models are constructed: the SimpleClassDiagram metametamodel, the PetriNet metamodel, and finally a PetriNet model. Together with the models, a simple conformance check algorithm is defined. This check can subsequently be executed on each of the exported models, to check whether or not they comply to their metamodel.

A model consists of a name, a colon, the name of the type, and the type mapping between parentheses. The type mapping should be a dictionary (or an empty node), which will be augmented with the typing information. The type mapping of multiple models can be stored in a single dictionary, though only a single type per model is allowed. In the modelling language, the first part constitutes the type of the element, which will be looked up in the specified metamodel. The second part is the identifier, which will be stored in the model for future referencing.

Listing 9.2: Simple modelling hierarchy, stored in `models.al`.

```
include "primitives.al"

Element typing_scd
typing_scd = create_node()

SimpleClassDiagram : SimpleClassDiagram (typing_scd) {
    Class Class()
    Type Type(Type)
    Type String(String)
    Attribute_ Attribute_ (Class, Type)
    Attribute_ AttributeAttrs (Attribute_, Type)
    Attribute_ Attribute (Class, Type)
    AttributeAttrs Name (Attribute, String)
    Association Association (Class, Class)
    Association Inheritance (Class, Class)
    Inheritance (Association, Class)
    Inheritance (Attribute_, Class)
}

PetriNet : SimpleClassDiagram (typing_scd) {
    Class Place()
        Attribute Place.tokens = Integer
            Name = "tokens"

    Class Transition()

    Association T2P (Transition, Place)
        Attribute T2P.weight = Integer
            Name = "weight"

    Association P2T (Place, Transition)
        Attribute P2T.weight = Integer
            Name = "weight"
}

my_net : PetriNet (typing_scd) {
    Place p1()
```

```
        Place.tokens = 3

    Place p2()
        Place.tokens = 0

    Transition t1()

    P2T (p1, t1)
        P2T.weight = 1

    T2P (t1, p2)
        T2P.weight = 2
}

export_node("metamodels/simpleclassdiagram", SimpleClassDiagram)
export_node("metamodels/petrinet", PetriNet)
export_node("models/my_net", my_net)
```

Listing 9.3: Conformance service, stored in `conformance.al`.

```
include "models.al"
include "conformance_scd.al"

while (True):
    output(conforms(input()))
```

10 Conclusion

In this paper, we described the Modelverse: a self-describable multi-paradigm modelling tool. Several axioms were presented, which served as guidelines while making decisions on the specification of the models. Our architecture was briefly presented, showing the distinction between the Interface (MvI), Kernel (MvK), and State (MvS).

We presented a model of the Modelverse, which defines how an implementation has to behave. The model covers both the way data is represented (in the MvS), and the semantics of its action language constructs (in the MvK).

Concerning data representation, we leave open how the graph could be physically implemented. This allows for a variety of implementations, allowing the developer to choose between available technologies. And as all implementations will be interoperable, users can try out different implementations and check whether it better matches with their goals.

Concerning the action language, we described the execution context representation, and how language primitives modify this execution context. This needs to be explicitly specified if multiple tools need to interoperate on the same piece of execution data. For example, an external debugger can now access all internal execution data, as its representation has been specified. For performance, we allow implementations to ignore updates to the execution context, allowing for optimized execution or primitive operations. This allows users to achieve higher efficiency, for example through compiled functions, although limiting debugability.

Tools can create and use additional elements in the execution context, which can be interpreted by compatible tools. However, tools have no obligation to support all these additional elements. An example is additional debugging information, such as tracing information.

By splitting up the components of the Modelverse, and requiring that all parts need to be explicitly modelled, we arrived at different notions of conformance. We distinguished between a conformance closer to the physical level (conformance$_\perp$), and a linguistic type of conformance closer to the user level (conformance$_L$). Whereas the physical notion allows users to circumvent strict metamodelling, by switching to a graph representation, linguistic conformance allows the MvK, and ultimately the user through the MvI, to reason about the model in a level that is close to the problem domain.

In future work, we will create a reference implementation of this specification. Apart from the reference implementation, multiple variations of components will be created, each with a different goal.

After the creation of the reference implementation, the implementation will be scaled up to a distributed and parallel version.

Multiple Modelverse Interfaces will also be created, each with a different kind of user in mind. First, a textual HUTN interface will be created. Afterwards, a graphical tool will be created.

Our different notions of conformance will also be further extended with the introduction of ontological conformance. This would allow us to have three different kinds of mappings: physical, linguistic, and ontological [26].

Bibliography

[1] Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation: Transactions of the Society for Modeling and Simulation International*, 80(9):433–450, 2004. Special Issue: Grand Challenges for Modeling and Simulation.

[2] Simon Van Mierlo, Bruno Barroca, Hans Vangheluwe, Eugene Syriani, and Thomas Kühne. Multi-level modelling in the modelverse. In *MULTI 2014 Multi-Level Modelling Workshop Proceedings*, pages 83–92, 2014.

[3] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan 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*, OOPSLA '97, pages 318–326, New York, NY, USA, 1997. ACM.

[4] Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[5] Tony Clark, Cesar Gonzalez-Perez, and Brian Henderson-Sellers. A Foundation for Multi-Level Modelling. In *MULTI 2014 Multi-Level Modelling Workshop Proceedings*, pages 43–52, 2014.

[6] Semantics of a Foundational Subset for Executable UML Models (FUML). http://www.omg.org/spec/FUML/, 2013.

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

[8] Frédéric Jouault, Massimo Tisi, and Jérôme Delatour. fuml as an assembly language for MDA. In *6th International Workshop on Modeling in Software Engineering, MiSE 2014, Hyderabad, India, June 2-3, 2014*, pages 61–64, 2014.

[9] Patrick Neubauer, Tanja Mayerhofer, and Gerti Kappel. Towards integrating modeling and programming languages: The case of UML and Java. Proceedings of the 2nd International Workshop on the Globalization of Modeling Languages, pages 23–32, 2014.

[10] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs Based on fUML. In Martin Erwig, RichardF. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 56–75. Springer International Publishing, 2013.

[11] Gergely Déva, Gábor Ferenc Kovács, and Ádám Ancsin. Textual, executable, translatable UML. 14th International Workshop on OCL and Textual Modeling Applications and Case Studies, pages 3–12. 2014.

[12] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Object Language (EOL). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg, 2006.

[13] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-oriented Meta-languages. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 264–278, Berlin, Heidelberg, 2005. Springer-Verlag.

[14] Ed Seidewitz. UML with meaning: executable modeling in foundational UML and the alf action language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 61–68, 2014.

[15] Daniel Balasubramanian, Tihamer Levendovszky, Abhishek Dubey, and Gabor Karsai. Taming multi-paradigm integration in a software architecture description language. In *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014.*, pages 67–76, 2014.

[16] Zoltán Micskei, Raimund-Andreas Konnerth, Benedek Horváth, Oszkár Semeráth, András Vörös, and Dániel Varró. On open source tools for behavioral modeling and analysis with fuml and alf. In *Proceedings of the 1st Workshop on Open Source Software for Model Driven Engineering co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, OSS4MDE@MoDELS 2014, Valencia, Spain, September 28, 2014.*, pages 31–41, 2014.

[17] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A runtime model for fuml. In *Proceedings of the 7th Workshop on Models@Run.Time*, MRT '12, pages 53–58, New York, NY, USA, 2012. ACM.

[18] Erwan Bousse, Benoît Combemale, and Benoit Baudry. Towards scalable multidimensional execution traces for xdsmls. In *Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation co-located with 17th International Conference on Model Driven Engineering Languages and Systems, MoDeVVa@MODELS 2014, Valencia, Spain, September 30, 2014.*, pages 13–18, 2014.

[19] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 — The Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer Berlin Heidelberg, 2000.

[20] Miklós Maróti, Róbert Kereskényi, Tamás Kecskés, Péter Völgyesi, and Ákos Lédeczi. Online Collaborative Environment for Designing Complex Computational Systems. *Procedia Computer Science*, 29(0):2432 – 2441, 2014. 2014 International Conference on Computational Science.

[21] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. AToMPM: A Web-based Modeling Environment. In *MODELS'13 Demonstrations*, 2013.

[22] Colin Atkinson and Ralph Gerbig. Melanie: Multi-level Modeling and Ontology Engineering Environment. In *Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 7:1–7:2, Innsbruck, Austria, 2012. ACM.

[23] Juan de Lara and Esther Guerra. Deep Meta-Modelling with MetaDepth. In *Proceedings of TOOLS, Lecture Notes in Computer Science vol. 6141*, pages 1–20. Springer, 2010.

[24] Levi Lucio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. 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*, pages 182–202. Springer, 2013.

[25] Eugene Syriani, Hans Vangheluwe, and Amr Al Mallah. Modelling and simulation-based design of a distributed DEVS simulator. In *Proceedings of the Winter Simulation Conference*, pages 3007–3021, 2011.

[26] Bruno Barroca, Thomas Kühne, and Hans Vangheluwe. Integrating language and ontology engineering. In *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014.*, pages 77–86, 2014.