Modelverse specification

Yentl Van Tendeloo
Bruno Barroca
Simon Van Mierlo
Hans Vangheluwe

August 31, 2016

Abstract

In this technical report, we present the specification of the Modelverse, a self-describable environment for multi-paradigm modelling. This specification defines how all hardcoded aspects of the Modelverse are defined, necessary to create a compliant Modelverse implementation. Apart from the core specification of the Modelverse, we also present the standardized API used between the different components. We do not commit ourselves to a single language, as this specification can be implemented in any language.

Contents

1 Introduction

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 can have any kind of front-end to 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 concepts such as conformance relations and model-management operations. 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 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.

In the remainder of this technical report, we present our requirements to the Modelverse in the form of axioms to which we will back-reference (Chapter 2). Chapter **??** presents the specification of the Modelverse State in a language-independent way. Chapter **??** presents the specification of the Modelverse Kernel. All communication between the intermediate layers is presented in Chapter **??**. Finally, Chapter **??** gives some practical information about running the Modelverse.

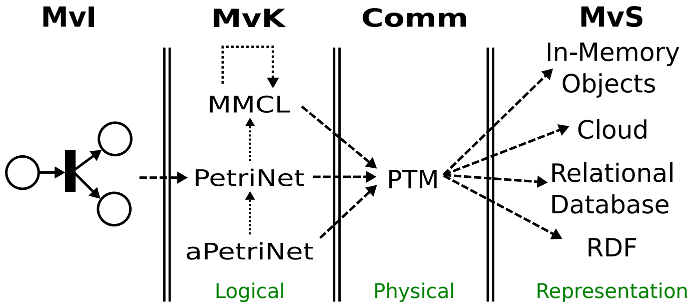


Figure 1.1: Overview of the Modelverse architecture

2 Axioms

NOTE: this section is still a work in progress.

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.

2.1 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.

2.2 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.

2.3 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 [1], . . .

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 [2], 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.

2.4 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.

2.5 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 [**?**].

2.6 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 [3], 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.

2.7 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.

2.8 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.

2.9 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.

2.10 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*.

2.11 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.

2.12 Interconnections

All of these axioms are related in some way, as the graph in Figure 2.1 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.

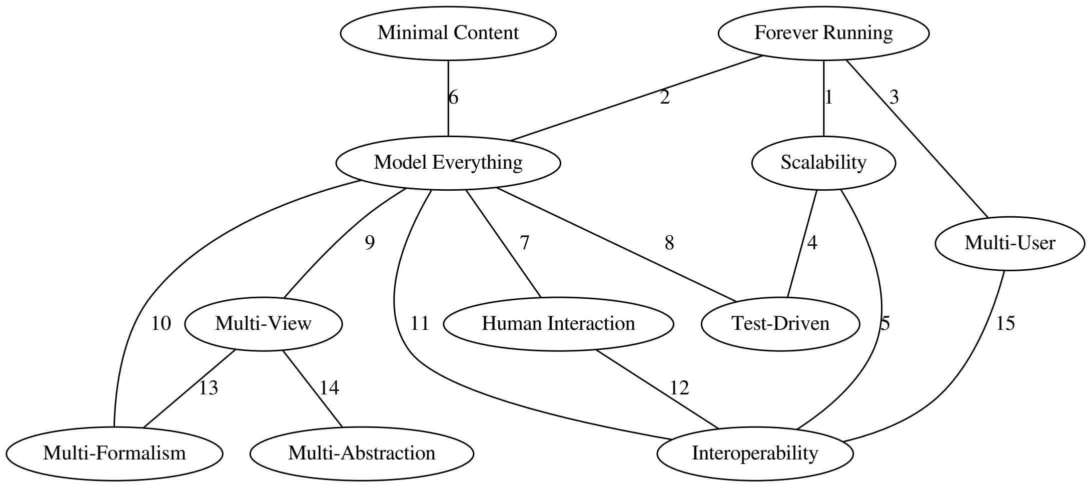


Figure 2.1: Overview of relations between all axioms

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.

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.

3 Modelverse State

We start our specification with the Modelverse State (MvS). The MvS maps the Physical Type Model (PTM) 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.

3.1 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, having a close 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).

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

$\mathbb{U}$ 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 be rounded towards the closest value in this format;
- **String** ($\mathbb{S}$) as the set of all ordered combinations of ASCII characters;
- **Boolean** ($\mathbb{B}$) as either True or False;
- **Action** ($\mathbb{A}$) as an action language construct, used to define Modelverse semantics. This is any of $\{If, While, Assign, Call, Break, Continue, Return, Resolve, Access, Constant, Declare, Global, Input, Output\}$.

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.

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

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).

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

3.2 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.

3.2.1 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).

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

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.

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

3.2.2 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.

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

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.

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

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

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. If the specified entry is not found in the dictionary, an error is raised.

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. Similarly, multiple outgoing edges might exist with the same label added to them, resulting in ambiguity. For performance reasons, however, the result will be non-deterministic.

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

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.

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

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

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

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

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

3.2.3 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 [4]. 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.

3.2.4 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.

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

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.

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

4
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.

4.1 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.

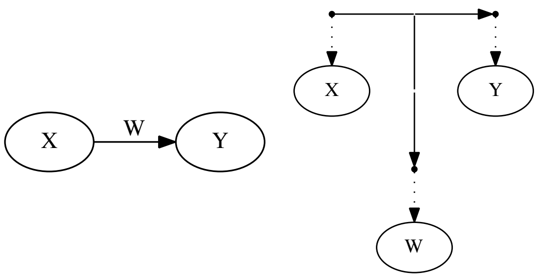
For each transformation rule, it is possible to decompose it in four distinct (sequentially ordered) components. The first two are read operations, which are used for the matching, and the last two are create and delete operations, which are used for the rewriting.

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 4.1: 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. 4.1a, equivalent to Fig. 4.1b, 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 4.2. First, the *success* status code is stored in s (equation 4.1), 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 4.2, followed by the negative read operation at equation 4.6. Now that all nodes and edges are bound, the create operation creates the necessary links starting from equation 4.7. From equation 4.10 to the end, operations try to match the edge to delete at a finer granularity and delete it in equation 4.18.

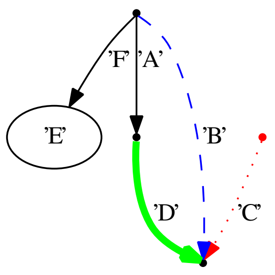


Figure 4.2: Example graph transformation which is expanded

$$s = (100, \_) \tag{4.1}$$
$$R_{dict}(G, a, ”A”) = (b, s) \tag{4.2}$$
$$R_{dict}(G, a, ”B”) = (c, s) \tag{4.3}$$
$$R_{dict}(G, a, ”F”) = (e, s) \tag{4.4}$$
$$R_V(G, e) = (”E”, s) \tag{4.5}$$
$$\nexists d : R_{dict}(G, d, ”C”) = (c, s) \tag{4.6}$$
$$C_{NV}(G, ”D”) = (G', f, s) \tag{4.7}$$
$$C_E(G', b, c) = (G'', g, s) \tag{4.8}$$
$$C_E(G'', g, f) = (G''', h, s) \tag{4.9}$$
$$\langle V_1, s\rangle = R_O(G''', a, s) \tag{4.10}$$
$$\langle V_2, s\rangle = R_I(G''', c, s) \tag{4.11}$$
$$i \in V_1 \tag{4.12}$$
$$i \in V_2 \tag{4.13}$$
$$\langle V_3, s\rangle = R_O(G''', i, s) \tag{4.14}$$
$$j \in V_3 \tag{4.15}$$
$$R_E(G''', j) = ((i, k), s) \tag{4.16}$$
$$R_V(G''', k) = (”B”, s) \tag{4.17}$$
$$D_E(G''', i) = (G'''', s) \tag{4.18}$$

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

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.

4.1.1 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:

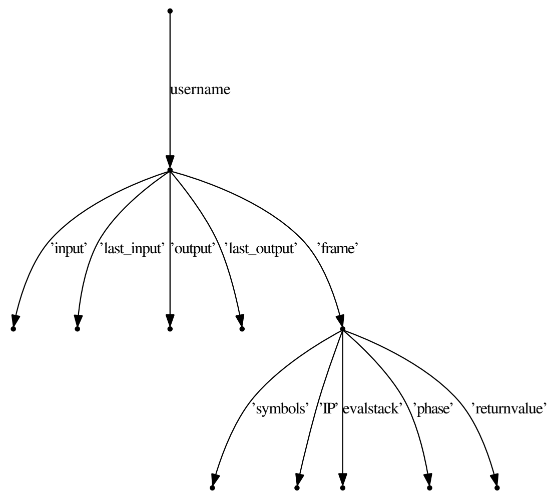


Figure 4.3: Graph to match as execution context. Some nodes might be identical.

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.

4.2 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. 4.3. 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.

4.3 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.

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 4.1: Outgoing link specification.

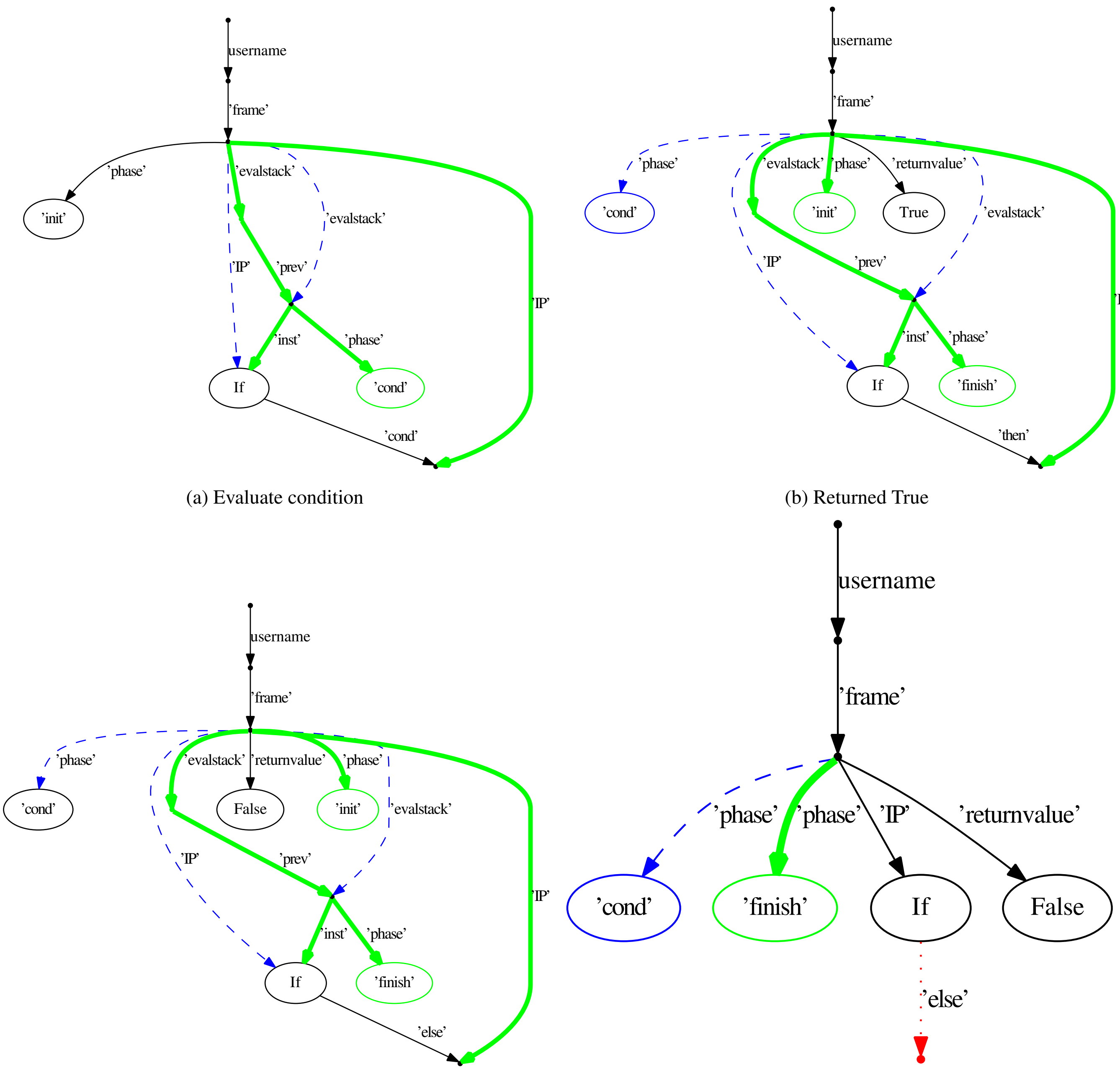
$$\frac{\begin{array}{c} G \xrightarrow{step_A} G' \vee \\ G \xrightarrow{step_B} G' \vee \\ \dots \end{array}}{G \xrightarrow{step} G'}$$

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) 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 4.1, 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 4.1.

4.3.1 If



(a) Evaluate condition

(b) Returned True

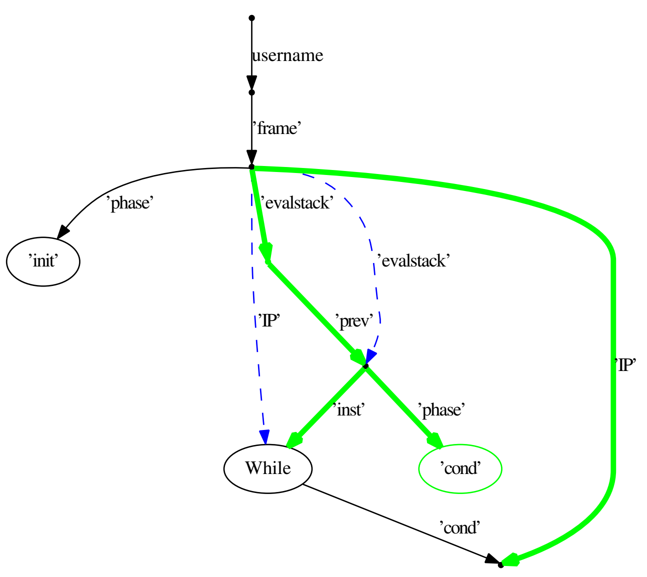
(c) Returned False and there is an 'else' block

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

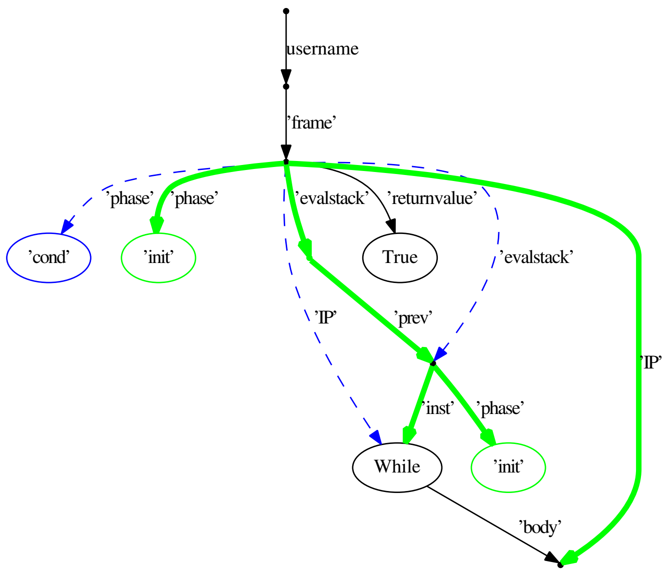
Figure 4.4: If branch rules

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 4.4a). 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 4.4b), 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 4.4c), it is executed, similar to the previous case. If it is False, but there is no *else* link (Figure 4.4d), the *If* is marked as completed immediately, without any subsequent actions.

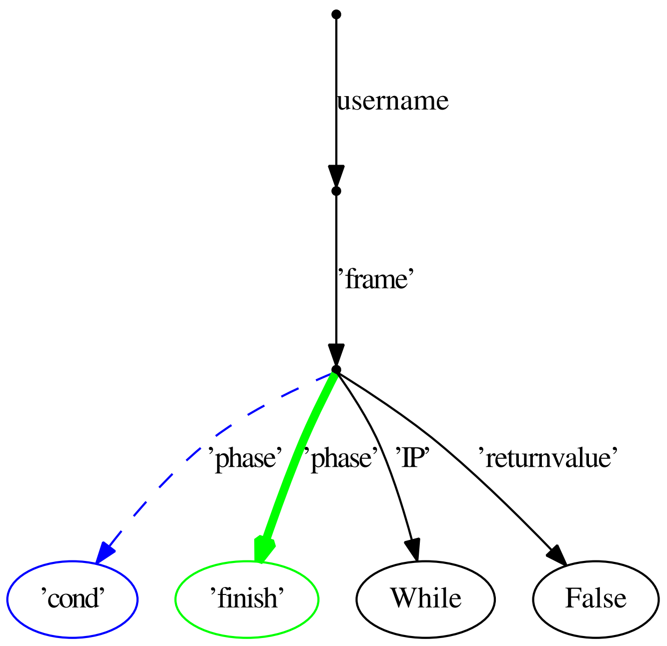
4.3.2 While



(a) Evaluate the condition of the While



(b) Condition was true



(c) Condition was false

Figure 4.5: 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 4.5a). 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 4.5b), 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 4.5c), the *While* is immediately marked as finished and the body is not executed.

4.3.3 Break

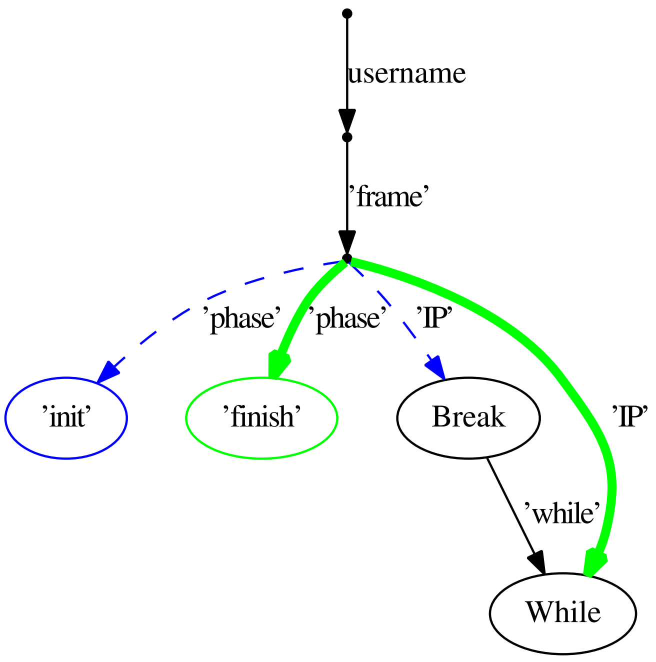


Figure 4.6: Break rule

The *Break* construct will move the instruction pointer back to the *While* construct it belongs to (Figure 4.6). 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.

4.3.4 Continue

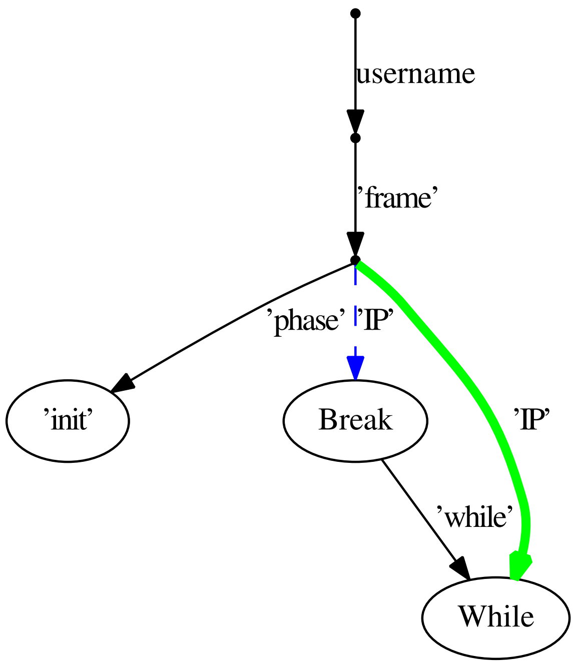
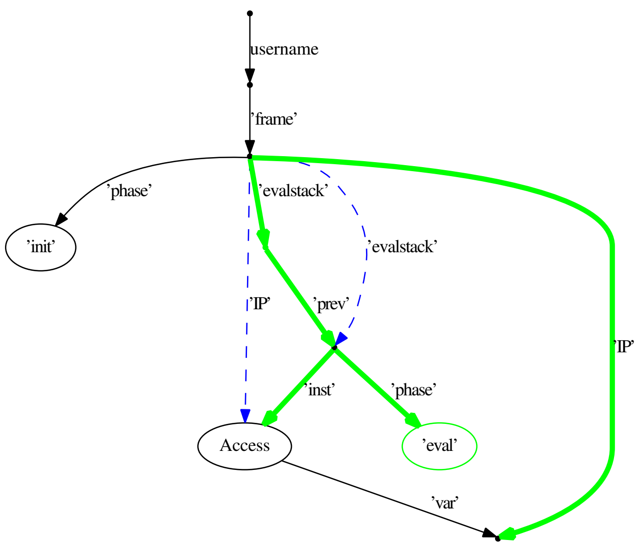


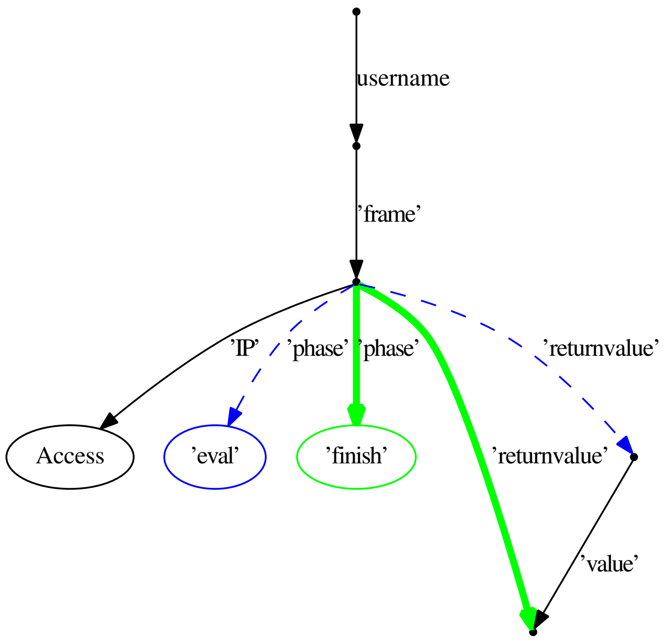
Figure 4.7: Continue rule

The *Continue* construct will move the instruction pointer back to the *While* construct to which it belongs (Figure 4.7). 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.

4.3.5 Access



(a) Evaluate variable to access

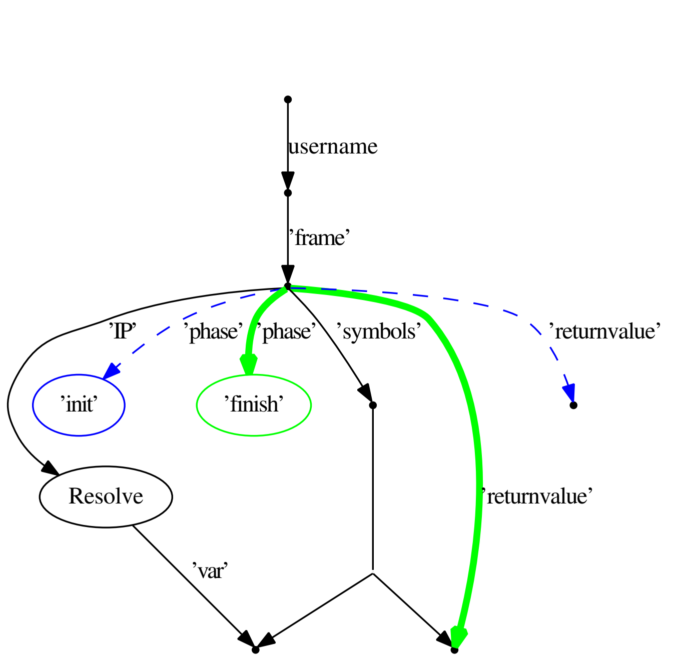


(b) Access evaluated variable

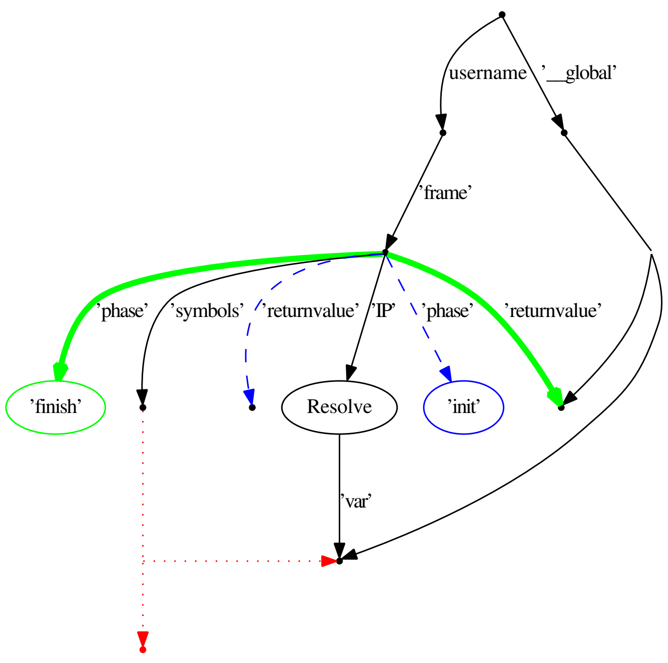
Figure 4.8: Variable dereference rules

The *Access* construct will move the instruction pointer to the variable which has to be resolved first (Figure 4.8a). 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 4.8b).

4.3.6 Resolve



(a) Access the variable from the local symbol table

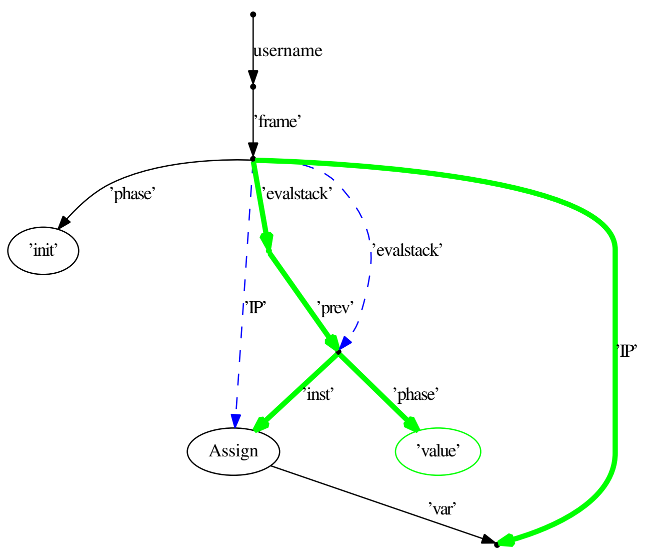


(b) Access the variable from the global symbol table

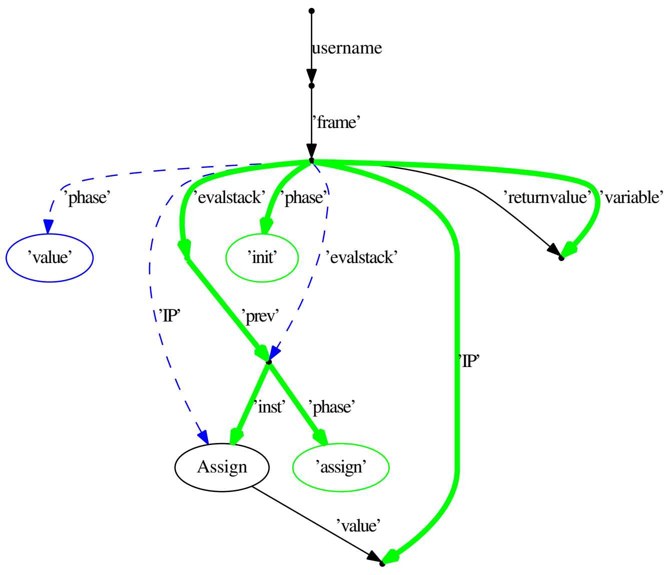
Figure 4.9: Resolution rules

With the *resolve* rule, a variable is looked up in either the local (Figure 4.9a) or global (Figure 4.9b) 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.

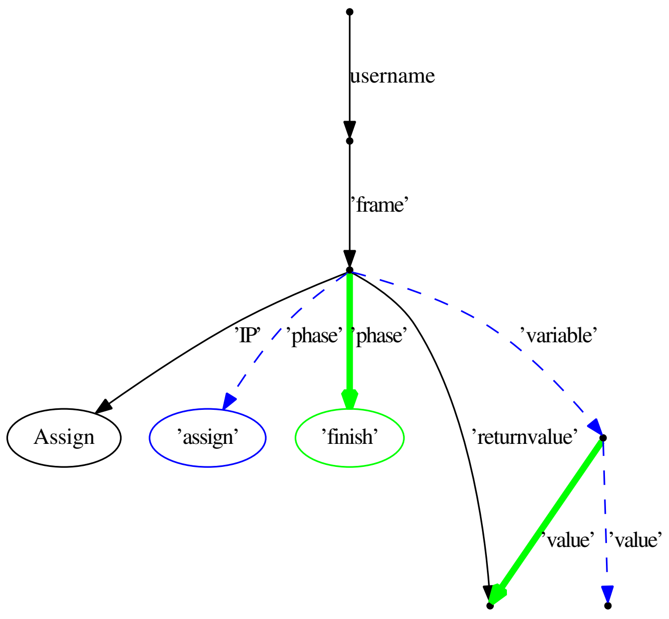
4.3.7 Assign



(a) Resolve the variable to assign to



(b) Evaluate the value to assign

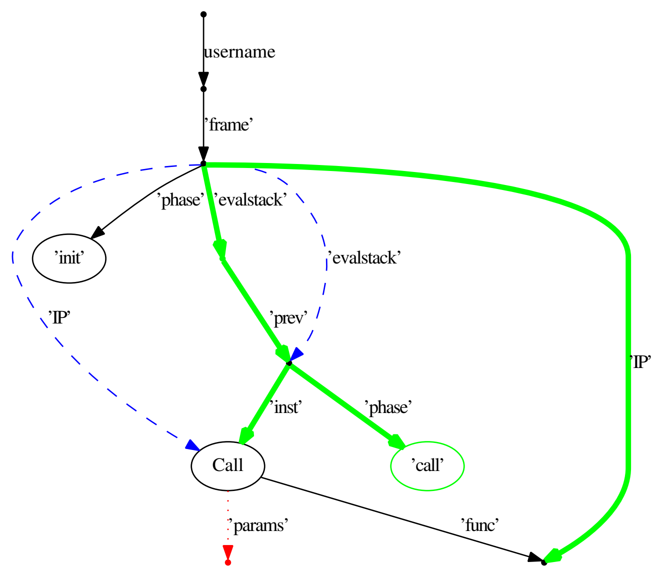


(c) Assign the value to the variable

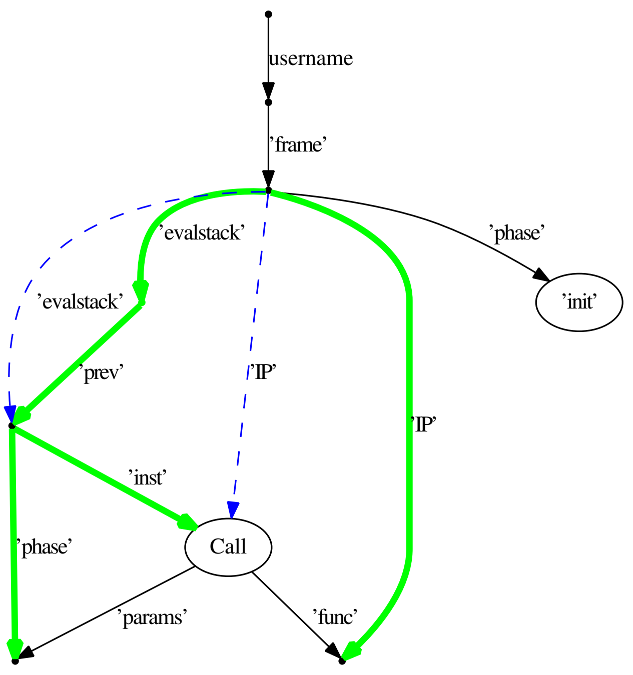
Figure 4.10: Assignment rules

The *Assign* rule will first evaluate the variable (Figure 4.10a), as it will first need to be resolved. After resolution (Figure 4.10b), 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 4.10c), the value link in the stored variable is changed to the evaluated value.

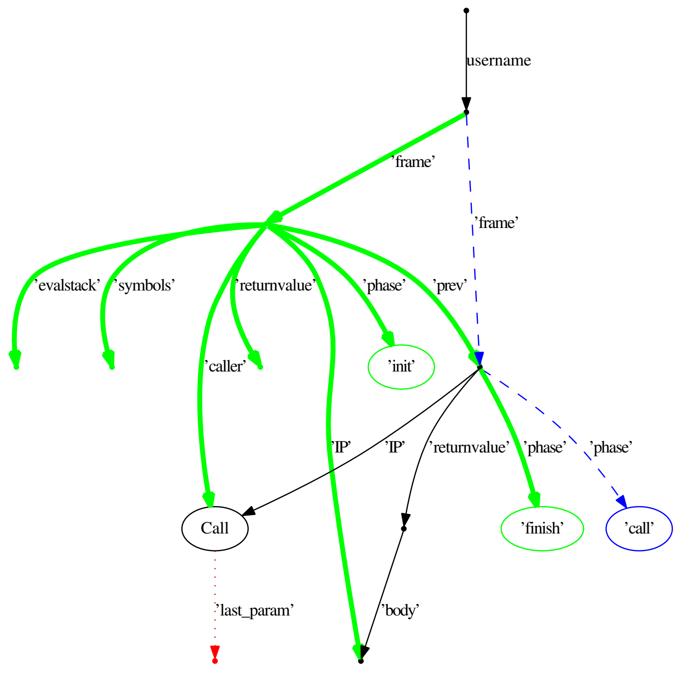
4.3.8 Function call



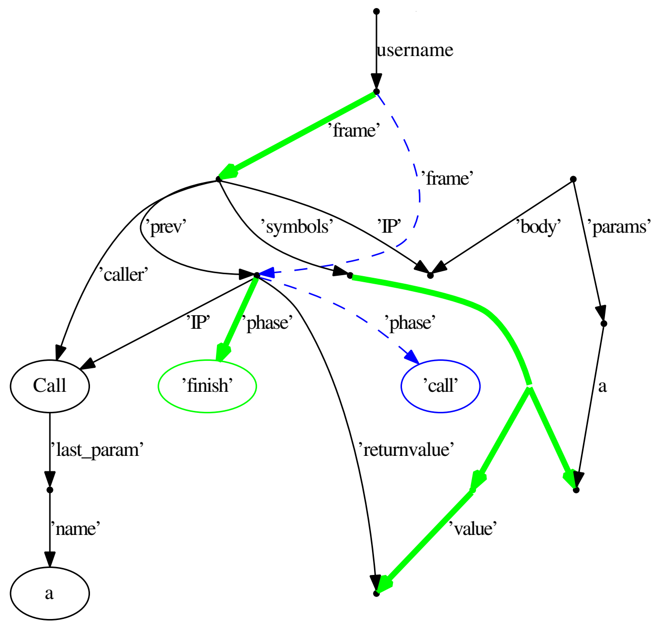
(a) Resolve function without parameters



(b) Resolve function with parameters



(c) Execute call with no parameters



(d) Execute call with parameters

Figure 4.11: 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 4.11a). After the function is resolved (Figure 4.11c), 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 4.11b), but instead of putting the *call* on the stack, the first parameter is used. Afterwards (Figure 4.12b), the stack is created for the resolved function,

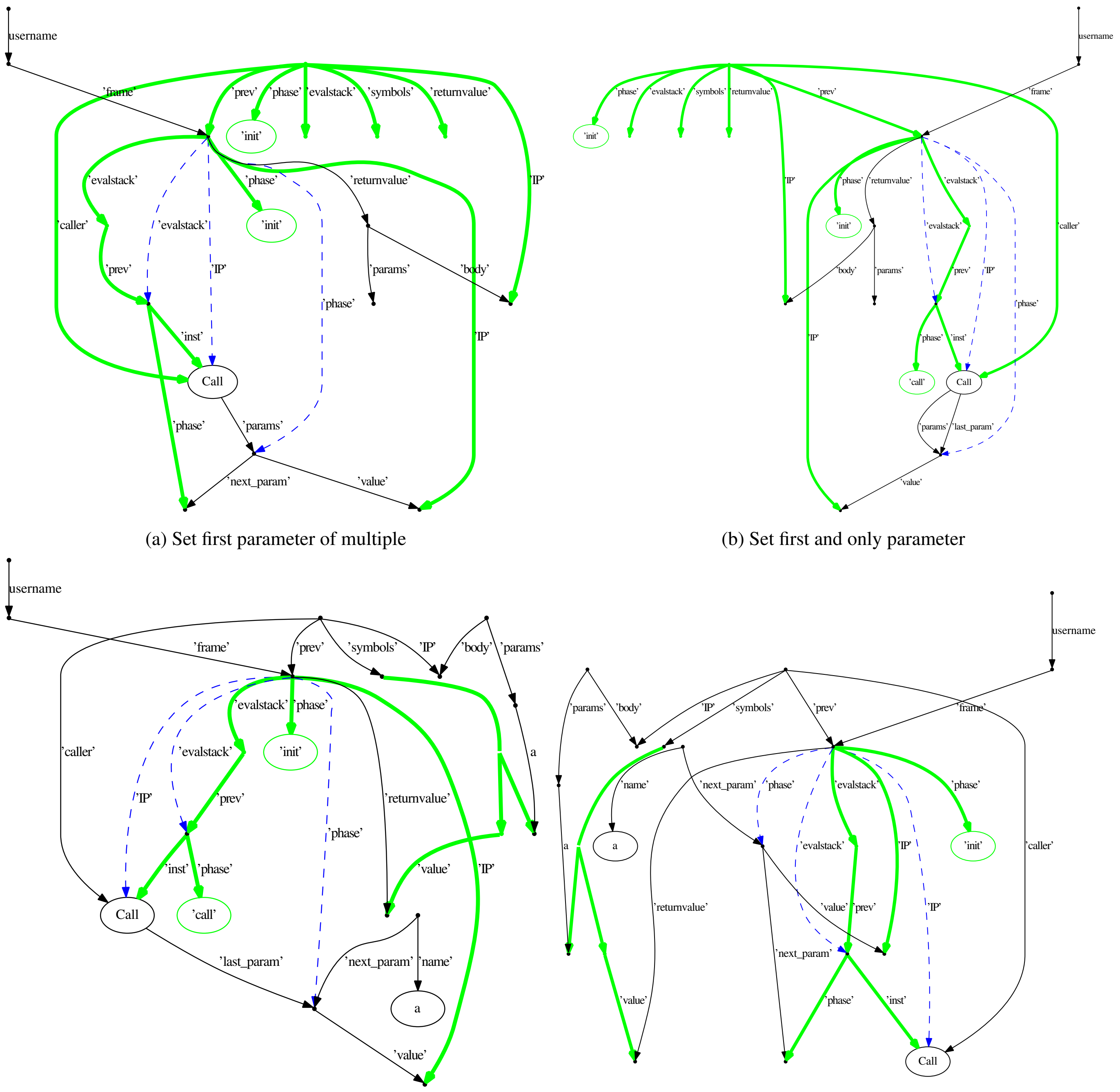
the instruction pointer is set to evaluate the argument, and the *call* is put on the stack. When the parameter is evaluated (Figure 4.11d), the result is put in the symbol table of the new execution frame and the new frame is made active.

3. **Two parameters**: similar to a single parameter, the first parameter is again put on the stack for after the function resolution (Figure 4.11c). When evaluating the first parameter (Figure 4.12a), 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 4.12c). Finally, the function is called as with only a single parameter (Figure 4.11d).

4. **More than two parameters**: similar to two parameters, but with an iteration rule (Figure 4.12d) 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

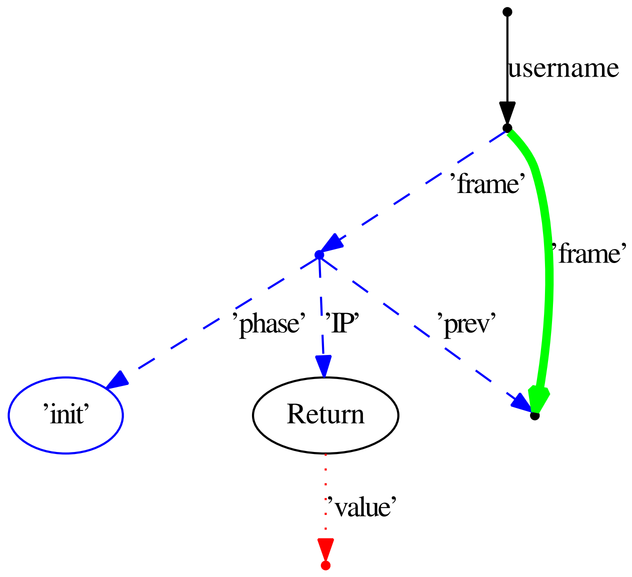
(b) Set first and only parameter

(c) Set last parameter of multiple

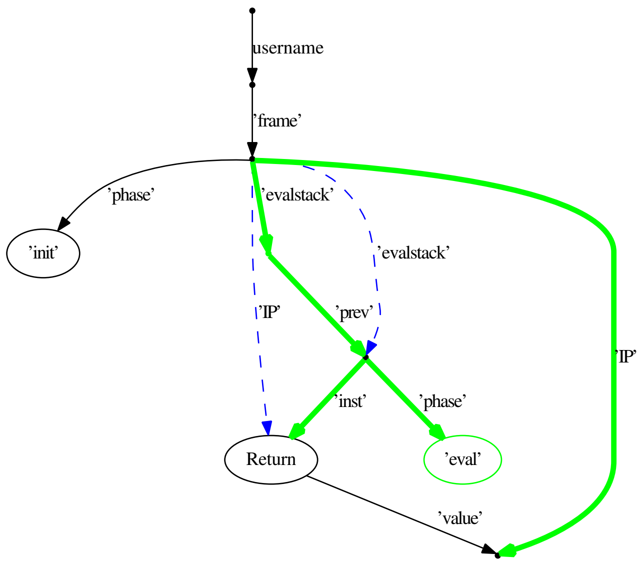
(d) Set next parameter

Figure 4.12: Function call rules for parameter evaluation

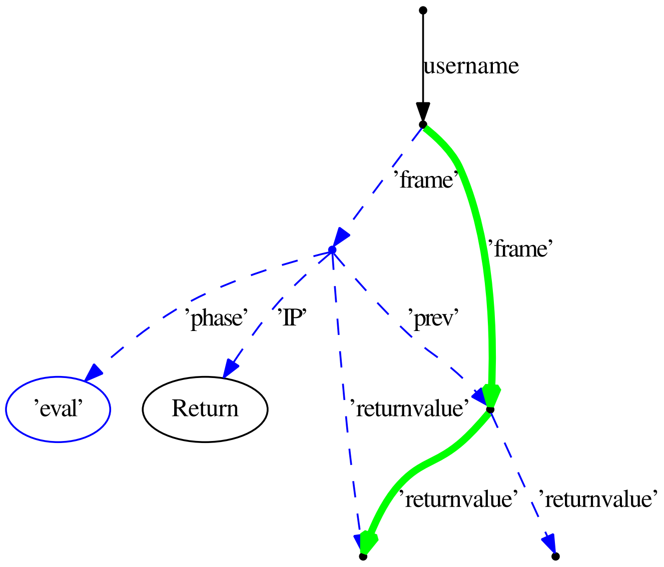
4.3.9 Return



(a) Return without value



(b) Evaluate the value of the return

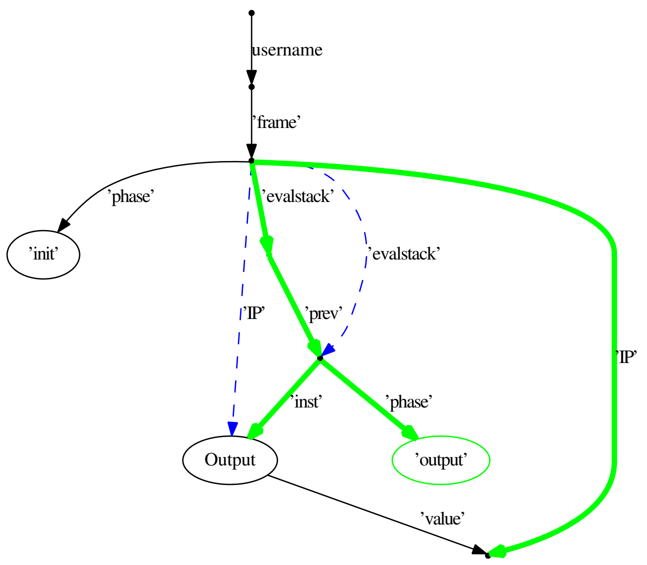


(c) Return with the evaluated value

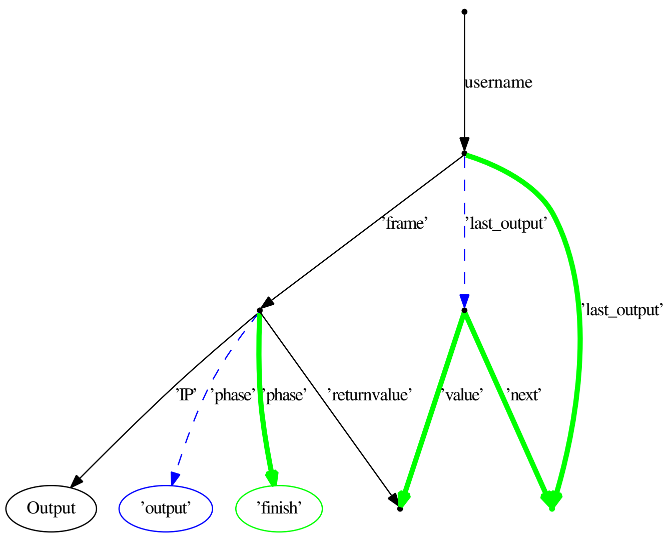
Figure 4.13: Return rules

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 4.13a), 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 4.13b), it is first evaluated by moving the instruction pointer there. After evaluation (Figure 4.13c), the evaluated value is stored in the returnvalue of the previous frame, and the current frame is deleted.

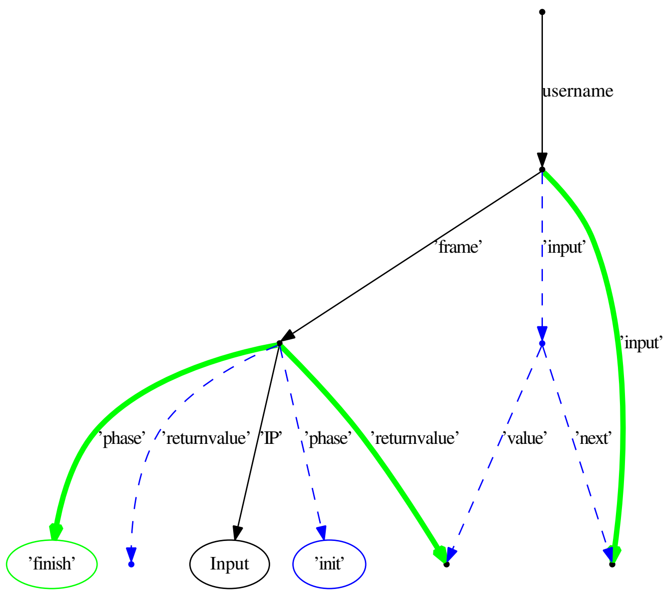
4.3.10 Input and Output



(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 4.14a), and afterwards it puts the returnvalue in the output queue (Figure 4.14b).

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.

4.3.11 Constant access

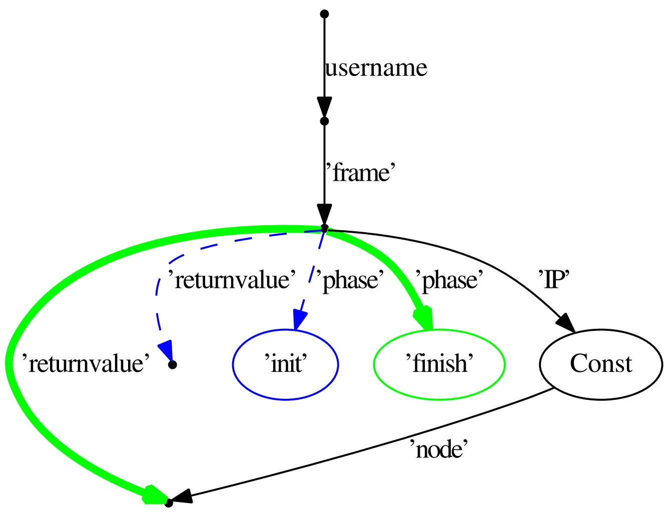
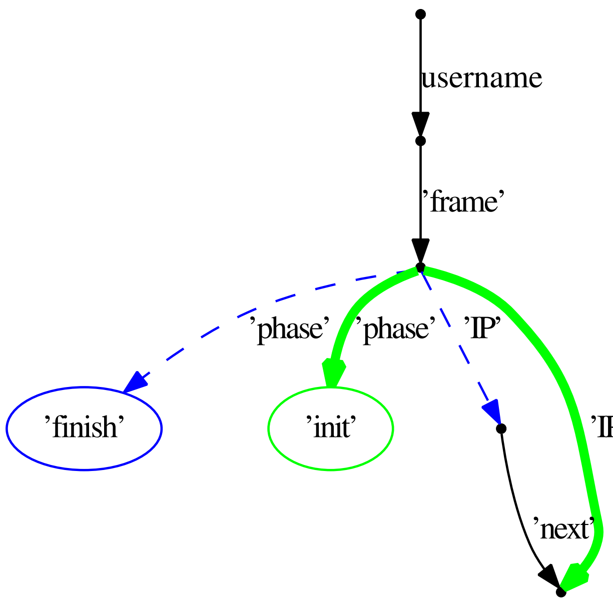


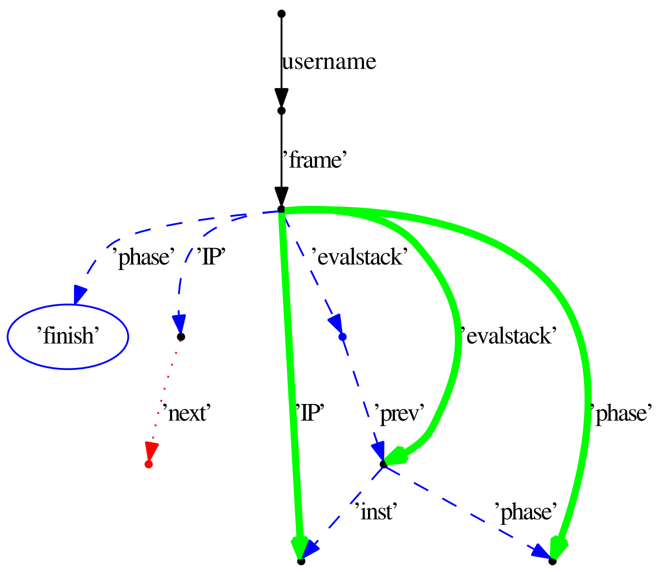
Figure 4.15: Constant access rule

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 4.15). The phase is also set to *finish*, to indicate termination of the construct.

4.3.12 Helper rules



(a) Progress to the next instruction



(b) Pop the next instruction from the stack

Figure 4.16: 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 4.16a), 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 4.16b), 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.

4.3.13 Declare

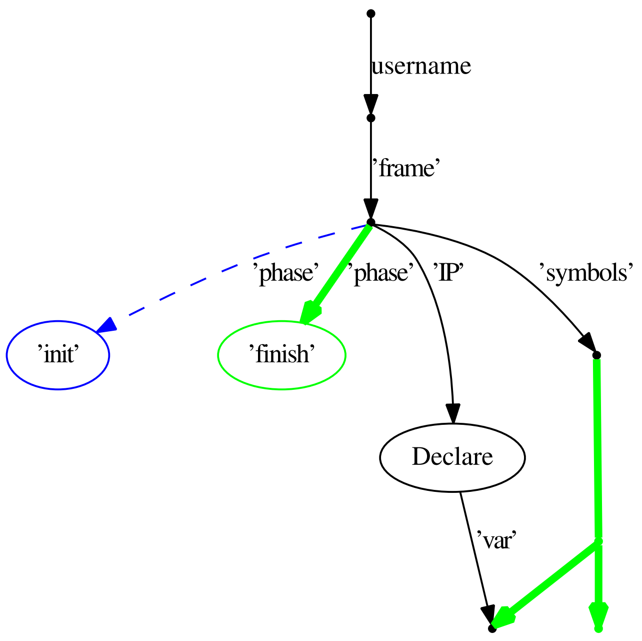


Figure 4.17: Declare instruction

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.

4.3.14 Global

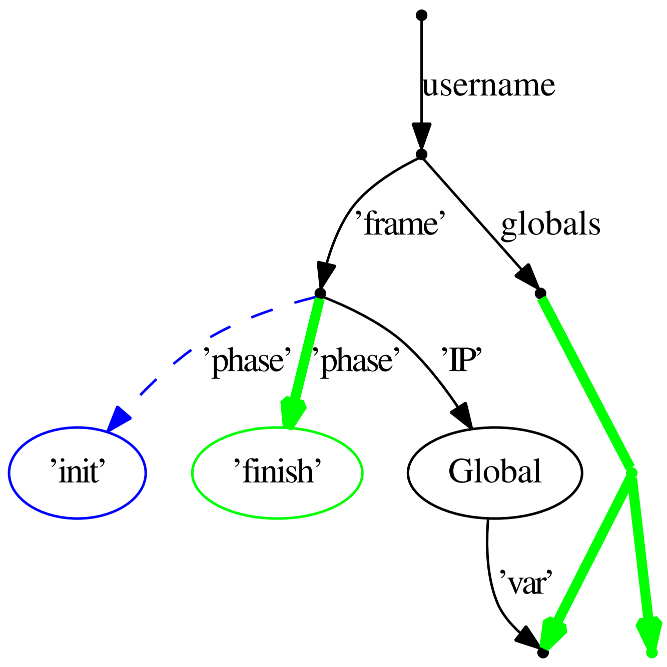


Figure 4.18: Global declare instruction

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.

4.4 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 [2], 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 4.2 and 4.3 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 4.2 and 4.3 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.
- **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_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_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_read	a: Element; b: Integer	c : Element	$c = a[b]$
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: Element	c : Element	$c = a[b.value]$
dict_read_edge	a: Element; b: Element	c : Element	$c = a[b]$
dict_read_node	a: Element; b: Element	c : Element	$c = a[b.id].edge$
dict_len	a: Element	b : Integer	$b = len(a)$
dict_in	a: Element; b: Element	c : Boolean	$c = bina$
dict_in_node	a: Element; b: Element	c : Boolean	$c = bina$
dict_keys	a: Element	b : Element	$b = a.keys()$
string_join	a: String; b: String	c : String	$c = a.b$
string_get	a: String; b: Integer	c : String	$c = a[b]$
string_split	a: String; b: String	c : Element	$c = a.split(b)$
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_remove_node	a: Element; b: Element	a : Element	$a.remove(b.id)$
set_in	a: Element; b: Element	c : Boolean	$c = bina$
value_eq	a: Element; b: Element	c : Bool	$c = a.value == b.value$
value_neq	a: Element; b: Element	c : Bool	$c = a.value \neq b.value$
element_eq	a: Element; b: Element	c : Bool	$c = a.id == b.id$
element_neq	a: Element; b: Element	c : Bool	$c = a.id \neq b.id$

Table 4.2: 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_a2s	a : Action	b : String	$b = str(a)$
cast_v2s	a : Element	b : String	$b = str(a.value)$
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
deserialize	a : String	b : Element	merge serialized graph with current and return initial node
log	a : String	a : String	print to console at Modelverse server
is_physical_int	a : Element	b : Boolean	$type(a.value) == integer$
is_physical_float	a : Element	b : Boolean	$type(a.value) == float$
is_physical_string	a : Element	b : Boolean	$type(a.value) == string$
is_physical_boolean	a : Element	b : Boolean	$type(a.value) == boolean$
is_physical_action	a : Element	b : Boolean	$type(a.value) == action$

Table 4.3: Lower-level primitive functions to implement. If a Value is taken or returned, this refers to the value of the returned node.

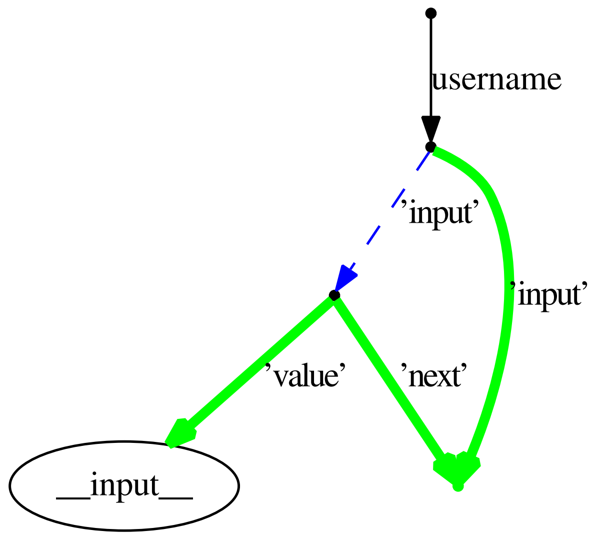
4.5 Interface

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.

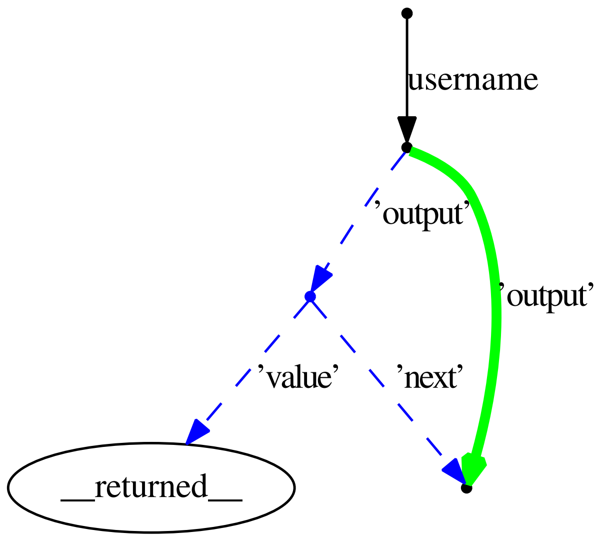
This interface is sufficient to execute all operations, as the input value can be any element in the modelverse, for example a function signature or name to resolve and subsequently execute. While this makes the interface very minimal, it pushes all API definitions to the MvK itself, thus explicitly 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 4.19: API rules

5 Network communication

Apart from the logical interface that is exposed, the physical interface should also be specified. This interface determines how the provided services can be invoked, possibly over the network. Since the Modelverse components are defined as seperate modules, with a standardized API, multiple back-ends can be created to couple them. Depending on the used back-end, the physical interface will differ. We will implement a simple back-end that makes the Modelverse state run as a service, exposed to the network. This is done with the XML/HTTP back-end, described below.

While XML/HTTP is not the most efficient protocol for this data exchange, it has the most wide portability and is not blocked by most firewalls. It also shows that our approach works with even a minimal communication protocol. It is possible to use other communication protocols as well. Even then, using XML/HTTP can still be used as a fallback method if other protocols are not supported.

5.1 Modelverse State

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 5.1 shows the mapping between the operation and the formalized function name.

Listing 5.1: Example request and reply

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

An example request, and corresponding reply, is shown in Listing 5.1, 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.

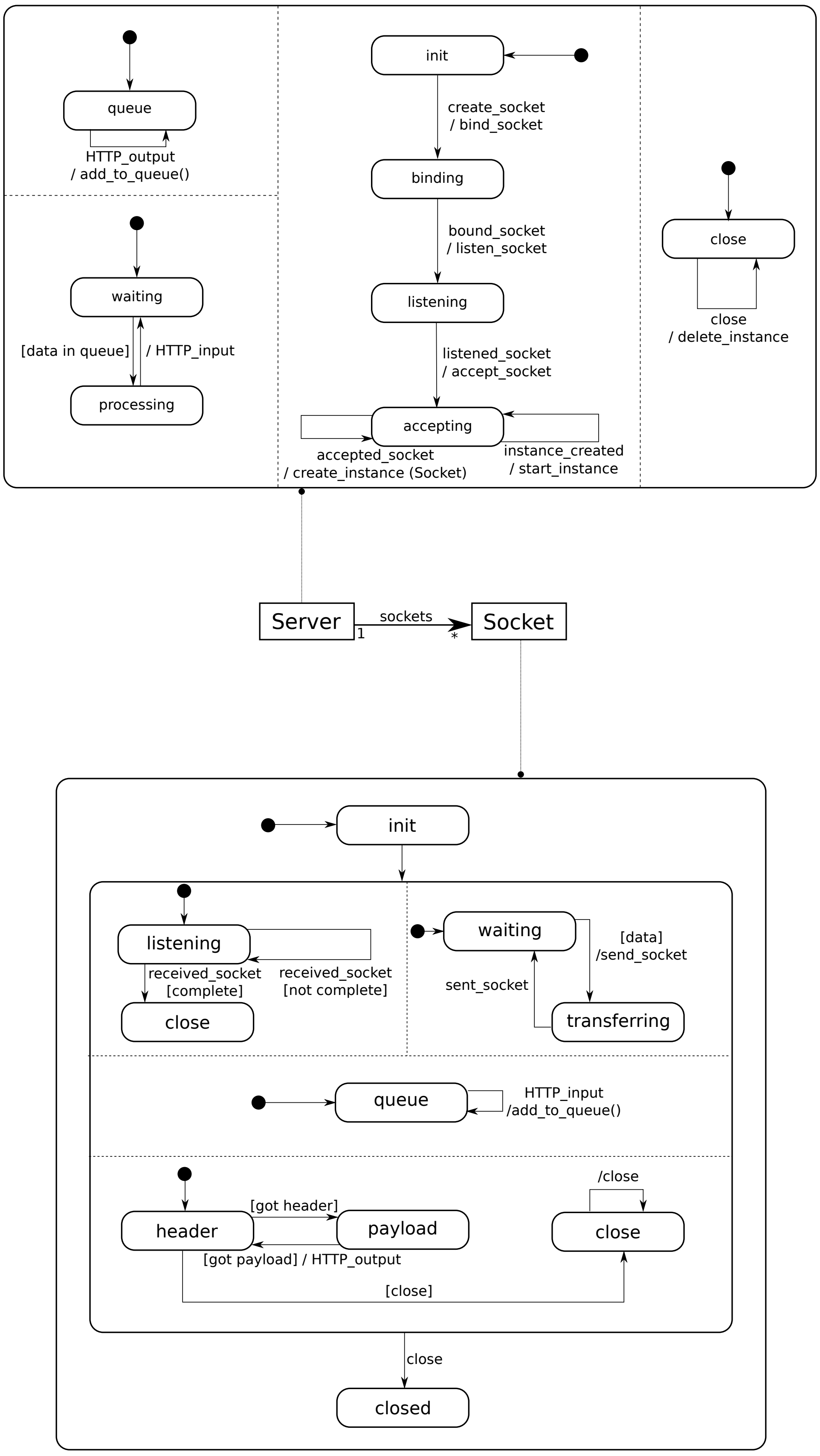


Figure 5.1: Modelverse State server statechart

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 5.1: Mapping between operations and formalized function name

5.1.1 Statechart

As the server component itself also contains non-trivial behaviour, it should just as well be modelled explicitly. For this purpose, SCCD [**?**] is ideally suited thanks to its support for reactive and timed behaviour, as well as dynamic structure. The SCCD structure is shown in Figure 5.1.

5.2 Modelverse Kernel

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:

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 5.2 and 5.3, for set_input, and Listing 5.4 and **??**, for get_output.

Listing 5.2: Example: create new user

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

Listing 5.3: 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 5.4: Example: read output value

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

5.2.1 Optimization

Whereas the communication between the MvK and MvS can be highly optimized and probably uses low-latency networks, the MvI and MvK most likely communicate over a high-latency and low-bandwidth connection. And since communication with the MvK can become relatively frequent, for example when executing a batch program, performance can be limited by the latency of the connection, and the overhead imposed for each small request. To circumvent this problem, the MvK accepts an additional parameter **data**, taking a JSON serialized list of JSON serialized element type and value tuples. The list is still executed in order. The reply will still just contain success, but will now be for all data simultaneously. An example is shown in Listing 5.5.

Listing 5.5: Example: create two new users simultaneously

```
Request: op=set_input&username=user_manager&data=[[\"V\", \"\\"user_1\\"\"], [\"V\", \"\\"user_2\\"\"]]
Reply:   id=100&value="success"
```

5.2.2 Statechart

Again, the MvK server has non-trivial behaviour and is therefore explicitly modelled using SCCD. This model can be seen in Figure 5.2. In contrast to the MvS statechart, which only modelled an HTTP server, the MvK models both an HTTP server and an HTTP client.

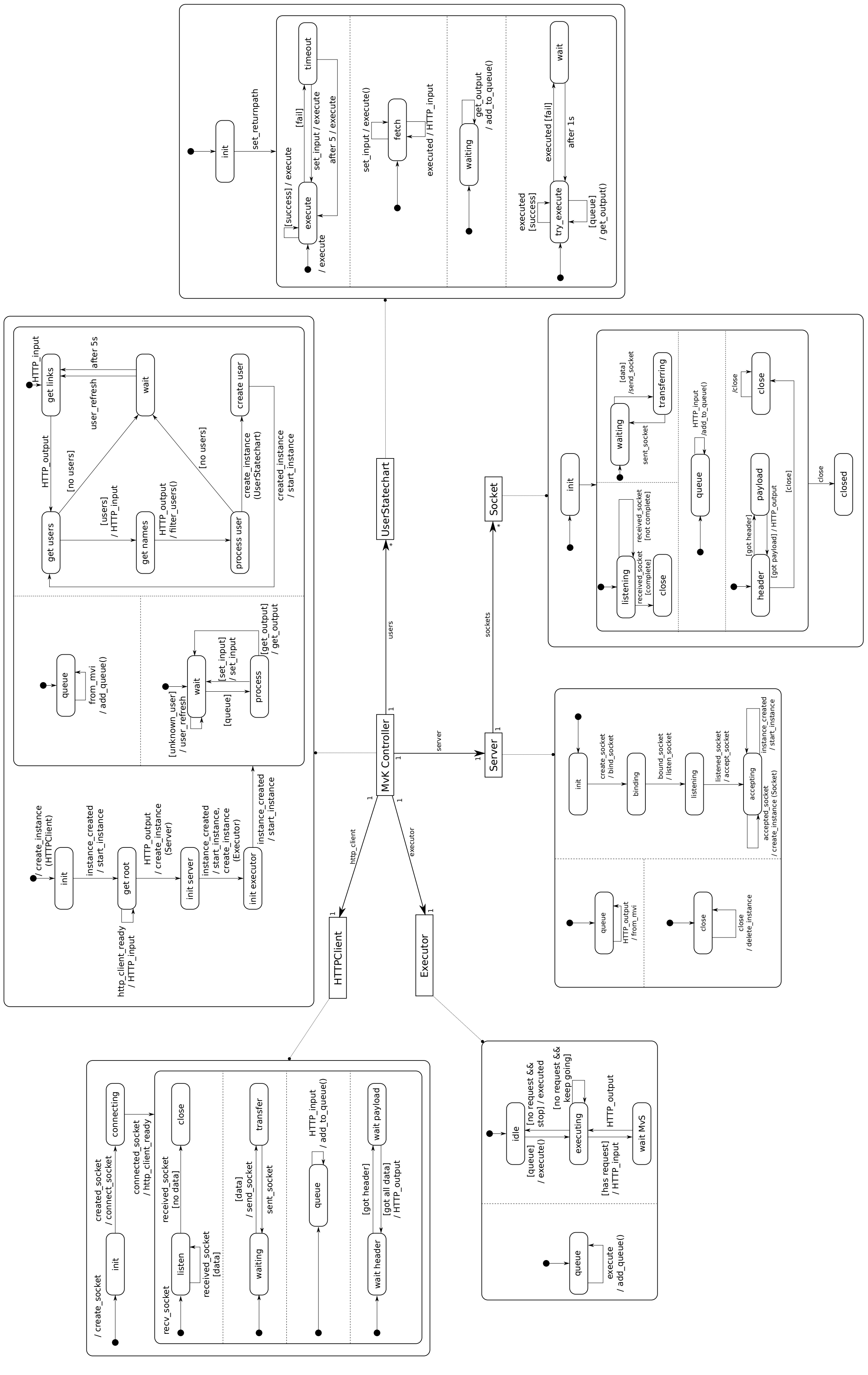


Figure 5.2: Modelverse Kernel server statechart

6
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.

6.1 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 written purely in Python and should therefore work on all platforms.

6.2 Test suite

Since the Modelverse project consists of several subprojects (Modelverse State, Modelverse Kernel, and Modelverse Interface), a script `run_tests.py` is provided which runs the tests of each component in order.

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.py` script.

6.3 Running the Modelverse

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

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.

6.4 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 `generate_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 is executed. If the provided code returns True, a deserialize call is invoked again, otherwise the user stops execution.

6.5 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="`.

6.6 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.

To compile and run your program, use the script `make_parallel.py`. It takes the address of the Modelverse, the username, and all files to compile and link together. For example, to execute `integration/code/factorial.alc`, execute: `python scripts/make_parallel.py http://localhost:8001 test bootstrap/*.alc integration/code/factorial.alc`. Note that this command also works on shells that don't do globbing (*i.e.*, expand the `*.alc`). There is also the slower, but more elegant, script `make_all.py`. `make_all.py` should be seen as the reference implementation, with `make_parallel.py` being an optimized version.

6.7 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.

6.7.1 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 6.1. 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.alh".
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 6.1: Example factorial service.

```
include "primitives.alh"

Integer function factorial(n : Integer):
    if(n < 1)):
        return 1
    else:
        return n * factorial(n - 1)

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

After compilation, as presented before, a user can 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.

7 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 [5].

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.